On the Formalization of Imperative Object-based Calculiin (Co)Inductive Type Theories

Alberto Ciaffaglione, Luigi Liquori, Marino Miculan

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On the Formalization of Imperative Object-based Calculi in (Co)Inductive Type Theories

Alberto Ciaffaglione — Luigi Liquori — Marino Miculan

N° 4812

April 2003

THÈME 2

ISSN 0249-6399

INRIA
LORRAINE

On the Formalization of Imperative Object-based Calculi in (Co)Inductive Type Theories

Alberto Ciaffaglione * , Luigi Liquori† , Marino Miculan ‡

Thème 2 — Génie logiciel
et calcul symbolique
Projet MIRÓ

Rapport de recherche n° 4812 — April 2003 — 37 pages

Abstract: In this paper, we study the formalization of Abadi and Cardelli's impς, a representative object-based calculus with types and side effects, in interactive proof assistants based on (Co)Inductive Type Theories, like Coq. In order to make the formal development of the theory of impς easier, we reformulate its static and dynamic semantics taking most advantage of the features provided by $CC^{(Co)Ind}$, the coinductive type theory underlying Coq. The new presentation is thus in the style of *Natural Deduction Semantics* (the counterpart in Natural Deduction style of Kahn's *Natural Semantics*), using *higher-order abstract syntax* and hypothetical-general premises *à la* Martin-Löf. Interestingly, for a significant fragment of impς we can even use *coinductive typing systems*, thus avoiding "store types" and leading to a substantial simplification of the proofs of key metaproperties, such as Subject Reduction.

The solutions we have devised in the encoding of and metareasoning on impς can be readily applied to other imperative calculi featuring similar issues.

Key-words: Interactive theorem proving, Logical foundations of programming, Program and system verification, Object-based calculi with side effects, Logical frameworks.

* DIMI, Università di Udine, Italy — ciaffagl@dimi.uniud.it and LORIA-INPL-ENSMNS, Nancy, France — ciaffagl@loria.fr
† INRIA-LORIA, France — Luigi.Liquori@inria.fr
‡ DIMI, Università di Udine, Italy — miculan@dimi.uniud.it

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 —Télécopie : +33 3 83 27 83 19

Un calcul à objets générique basé sur les systèmes de réécriture de termes à adresses

Résumé : Dans cet article, nous étudions la formalisation du calcul impς d'Abadi et de Cardelli, un calcul à objets purs avec des types et des effets de bord, avec l'aide des assistants à la preuve interactifs basés sur la théories des types Co-Inductive, comme `Coq`. Pour simplifier le développement formel de la théorie de impς, nous reformulons sa sémantique statique et dynamique en profitant de la plupart des dispositifs fournis par $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, i.e. la théorie des types coinductifs. La nouvelle présentation est ainsi dans un nouveau style appelé *Natural Deduction Semantics* (la contrepartie dans la déduction naturelle de la *Natural Semantics* à la Kahn), en utilisant la syntaxe d'ordre supérieure et des prémisses hypothétiques à la Martin-Löf.

Pour un fragment significatif de impς on peux utiliser la technique de coinduction pour simplifier la structure de "types mémoire", ce qui réduit significativement les démonstrations de métathéorie comme le théorème de réduction du sujet.

Mots-clés : Calculs basés sur les objets, objets partagés, objets mutables, objets cycliques, substitution explicite, réécriture de termes à adresses.

Introduction

In recent years, there has been a lot of effort in the formalization of class-based, object-oriented languages. The `Coq` system [22] has been used for formalizing the `JavaCard` Virtual Machine and studying formally the `JavaCard` Platform [4,3], and for checking the behaviour of a byte-code verifier for the JVM language [6]. PVS and Isabelle have been used for formalizing and certifying an executable bytecode verifier for a significant subset of JVM [24], for reasoning on Java programs with Hoare-style logics [21], and for applying translations of co-algebraic specifications [29] to programs in `JavaCard` [33] and `C++` [32].

In spite of this large effort on class-based languages, relatively little or no formal work has been done on *object-based* ones, like `Self` and `Obliq`, where there is no notion of "class" (though classes can be modeled by objects able to receive the message `new`, which corresponds to the creation of another object). This is due mainly to the fact that object-based languages are less used in common practice than class-based ones; despite this, they are simpler to implement and understand and can be used as common intermediate level for implementing interpreters and compilers for the latters. From a foundational point of view, indeed, most of the calculi introduced for the mathematical analysis of the object-oriented paradigm are object-based [1, 13]. Among the other calculi, Abadi and Cardelli's impς [1], is particularly representative: it features objects, methods, dynamic lookup, method override, types, subtypes, and, last but not least, imperative features. Clearly, all this makes impς and similar calculi quite complex, both at the syntactic and at the semantic level. impς features all the idiosyncrasies of functional languages with imperative features; moreover, the store model underlying the language allows for loops, thus making the typing system for values quite awkward. This level of complexity is reflected in developing metatheoretic properties; for instance, the fundamental *subject reduction* and *type soundness* for impς are much harder to state and prove than in the case of usual functional languages. It is clear that this situation can benefit from the use of *proof assistants*, where the theory of the object calculus can be formally represented in some metalanguage, the proofs can be checked and new, error-free proofs can be safely developed in interactive sessions. However, up to our knowledge there is no formalization of a object-based calculus like impς, yet. Therefore, the study of the formal definitions and implementations of object-based calculi becomes actual and challenging in the more general setting of *mechanized software verification and certification.*

This is indeed the aim of this work. In this paper we represent and reason on both static and dynamic aspects of impς in an interactive proof assistant, i.e. `Coq`. To this end we will use `Coq`'s specification language, the coinductive type theory CC$^{(Co)Ind}$, as a Logical Framework (`LF`). Following the encoding methodology of Logical Frameworks, we are forced to spell out in full detail all aspects of the calculus: this gives the possibility to identify and fix problematic issues which are skipped on the paper. Moreover, we can (re)formulate the object system taking full advantage of the definition and proof-theoretical principles provided by the Logical Framework. In particular most type theory-based `LF`s support *natural deduction*, *higher-order abstract syntax*, and even *coinductive* datatypes and predicates, as in the case of CC$^{(Co)Ind}$. The `LF` perspective may thus suggest alternative, and cleaner, definitions of the same systems.

Therefore, before implementing impς in `Coq` we reformulate its operational semantics and the typing system in the style of the *Natural Deduction Semantics* (`NDS`) [7, 25], the counterpart in Natural Deduction of Kahn's *Natural Semantics* (`NS`) [23, 11] adopted in [1]. In this setting, the handling of structures which obey a stack discipline (such as the *environments*) is delegated to the metalanguage, by means of hypothetical-general premises à la Martin-Löf. Hence, environments do not appear explicitly anymore in judgments and proofs, which become appreciably simpler.

Another key proof-theoretical innovation of our rephrasing of impς is the use of *coinductive* types and proof systems. This is motivated by observing that the proof of the Subject Reduction in [1] is quite involved mainly because the store may contain "pointer loops". Since loops have non-wellfounded nature, usual inductive arguments are invalidated and extra structures, the *store types*, have to be introduced and dealt with in already long and error-prone proofs. However, several today type theories and proof assistants provide *coinductive types*, which can be fruitfully used as the canonical way for dealing with circular, non-wellfounded objects. Therefore, we present

an original *coinductive* reformulation of the typing system for the fragment of $\mathsf{imp}\varsigma$ without *object override* (which we denote by $\mathsf{imp}\varsigma^{\mathrm{nov}}$), thus getting rid of the extra structure of store types and making the proof of Subject Reduction dramatically simpler.

Unfortunately, the same does not seem to be feasible in the case of the full $\mathsf{imp}\varsigma$, where we have to resort to store types and related typing system. Anyway, we remark that most of the development carried out for $\mathsf{imp}\varsigma^{\mathrm{nov}}$ can be readily recovered for $\mathsf{imp}\varsigma$, thus pointing out the *modularity* of our development.

Our effort is useful also from the point of view of Logical Frameworks. The theoretical development of LFs and their implementation will benefit from complex case studies like the present one, where we test the applicability of advanced encoding methodologies. In this perspective, our contribution can be considered pioneering in combining the higher-order approach with coinductive natural deduction style proof systems. The techniques we have developed in the encoding of and metareasoning on $\mathsf{imp}\varsigma$ can be reused for other imperative calculi featuring similar issues.

Synopsis. The paper is structured as follows. Section 1 gives a brief account of $\mathsf{imp}\varsigma$. In Section 2 we focus on the fragment $\mathsf{imp}\varsigma^{\mathrm{nov}}$, which is reformulated bearing in mind the proof-theoretical concepts provided by $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$. In Section 3 we scale up towards the full $\mathsf{imp}\varsigma$ calculus. The formalization in Coq of these systems, and the formal proofs of some of their key properties, are discussed in Sections 4 and 5 respectively. Conclusions and directions for future work are presented in Section 6. In Appendix A we recall briefly the Calculus of (Co)Inductive Constructions. Longer listings of Coq code are in Appendix B; longer proofs in Appendixes C and D.

1 Abadi and Cardelli's $\mathsf{imp}\varsigma$ Calculus

In this section, we introduce and survey the $\mathsf{imp}\varsigma$ calculus of Abadi and Cardelli [1, Chapter 10,11]: in a nutshell it is an imperative, calculus of objects forming the kernel of the Obliq [8] programming language. Its type inference system is sound in the sense that every well-typed object will not invoke methods not declared in its interface. The language of $\mathsf{imp}\varsigma$ is the following:

a, b	$::=$	x	variable	$a.l$	method invocation
		$[l_i = \varsigma(x_i)b_i]^{i\in I}$	object	$a.l \Leftarrow \varsigma(x)b$	method update
		$clone(a)$	cloning	$let\ x = a\ in\ b$	local declaration

- An *object* $[l_i = \varsigma(x_i)b_i]^{i\in I}$ is a collection of components $l_i = \varsigma(x_i)b_i$ for distinct method names l_i and associated methods $\varsigma(x_i)b_i$. The parameter x_i has to be bound to the method's *host* object, the object containing the given method. The order of the components does not matter. The calculus $\mathsf{imp}\varsigma$ dispenses with the fields, because the method suite is mutable; fields could be encoded using the *let* construct.

- *Method invocation* $a.l$, where the method named l in a is $\varsigma(x)b$, has the intent of executing the body b with the parameter x bound to the host object a, thus returning the result of the execution.

- *Method update* $a.l \Leftarrow \varsigma(x)b$ is a typical imperative operation: it replaces *in place* the method named l in a with $\varsigma(x)b$ and then returns the modified object.

- The *cloning* operation $clone(a)$ is characteristic of object-based languages: it produces a new object with the same labels of a, with each component sharing the methods of the corresponding component of a.

- The *let* construct $let\ x = a\ in\ b$ evaluates a term a, binds the result to a variable x and then evaluates a second term b with the variable x in the scope, thus permitting to have local definitions and to control the execution flow. For example, sequential evaluation is defined as $a; b \triangleq let\ x = a\ in\ b$, if $x \notin \mathsf{FV}(b)$.

$$\frac{}{\emptyset \vdash \diamond}\ (Store{-}\emptyset) \qquad \frac{\sigma \bullet S \vdash \diamond \quad \iota \notin \mathsf{Dom}(\sigma)}{\sigma, \iota \mapsto \langle \varsigma(x)b, S\rangle \vdash \diamond}\ (Store{-}\iota)$$

$$\frac{\sigma \vdash \diamond}{\sigma \bullet \emptyset \vdash \diamond}\ (Stack{-}\emptyset) \qquad \frac{\sigma \bullet S \vdash \diamond \quad \iota_i \in \mathsf{Dom}(\sigma) \quad x \notin \mathsf{Dom}(S) \quad \forall i \in I}{\sigma \bullet (S, x \mapsto [l_i = \iota_i]^{i\in I}) \vdash \diamond}\ (Stack{-}Var)$$

Figure 1: Rules for Store and Stack Auxiliary Judgments

1.1 Operational Semantics

The Natural Semantics (NS) *à la Kahn* is expressed by a reduction relation that relates a store σ, a stack S, a term a, a result v and another store σ', *i.e.* $\sigma \bullet S \vdash a \rightsquigarrow v \bullet \sigma'$. The intended meaning is that with the store σ and the stack S, the term a reduces to a result v, yielding an updated store σ' and leaving the stack S unchanged in the process. The entities used in the big-step semantics of impς are the following:

ι	$\in$	Nat	store location	S	$::=$	$x_i \mapsto v_i{}^{i\in I}$	stack
v	$::=$	$[l_i = \iota_i]^{i\in I}$	result	σ	$::=$	$\iota_i \mapsto \langle \varsigma(x_i)b_i, S_i\rangle^{i\in I}$	store

The global *store* is a function mapping locations to *method closures*. Closures (denoted by k) are pairs built of *methods* and *stacks*; stacks are used for the reduction of the corresponding method body: they associate variables with *object results*. Results are sequences of pairs: method labels together with store locations, one location for each object component. Notice that, in order to stay close to implementation techniques, there is no use of the notion of formal substitution. Moreover the operational semantics needs two more auxiliary judgments, namely $\sigma \vdash \diamond$, and $\sigma \bullet S \vdash \diamond$ both checking the well-formedness of stores and stacks, respectively. In the following, the notation $\iota_i \mapsto k_i^{i\in I}$ denotes the store that maps the location ι_i to the closure k_i, for $i \in I$; the store $s, \iota \mapsto k$ extends σ with k at ι (fresh) and $s.\iota_j \leftarrow k$ denotes the result of storing k in the location ι_j of σ. Unless not explicitly remarked, we assume all l_i, ι_i be distinct. Auxiliary judgments are presented in Figure 1.

- A well-formed store is built starting from the well-formed empty sequence and allocating a new closure pointed to by a fresh pointer, provided the closure contains a well-formed stack;
- The empty well-formed stack is built starting from a well-formed store;
- A well-formed stack can be enriched pushing on its top the association between a fresh variable and the result formed by pairs of distinct labels and distinct pointers referring to meaningful store locations.

Natural Semantics relates terms to results in stores; it is defined in Figure 2. In a nutshell:

- A *variable* reduces to the result it denotes in the current stack;
- An *object* reduces to a result consisting of a fresh collection of locations, such that the store is extended for associating the method closures to these locations;
- A *selection* operation first reduces its host object to a result, then activates the appropriate method closure;
- An *update* operation reduces its object and then updates the appropriate store location with a new method closure;
- A *cloning* operation reduces its object and then allocates a fresh collection of locations and associates them to the existing method closures from the object (deep cloning);
- A *let* construct reduces to the result of reducing its body in a stack extended with the bound variable associated to the result of its local term.

Finally, notice that an algorithm for reduction can be easily extracted from the rules: it would parallel standard implementations of objects.

$$\frac{\sigma \bullet (S', x \mapsto v, S'') \vdash \diamond}{\sigma \bullet (S', x \mapsto v, S'') \vdash x \rightsquigarrow v \bullet \sigma} \quad (Red{-}Var)$$

$$\frac{\sigma \bullet S \vdash \diamond \quad \iota_i \notin \mathsf{Dom}(\sigma) \quad \forall i \in I}{\sigma \bullet S \vdash [l_i = \varsigma(x_i)b_i]^{i\in I} \rightsquigarrow [l_i = \iota_i]^{i\in I} \bullet (\sigma, \iota_i \mapsto \langle \varsigma(x_i)b_i, S\rangle)^{i\in I}} \quad (Red{-}Obj)$$

$$\frac{\begin{array}{ll}\sigma'(\iota_j) = \langle \varsigma(x_j)b_j, S'\rangle & x_j \notin \mathsf{Dom}(S') \qquad j \in I \\ \sigma \bullet S \vdash a \rightsquigarrow [l_i = \iota_i]^{i\in I} \bullet \sigma' & \sigma' \bullet (S', x_j \mapsto [l_i = \iota_i]^{i\in I}) \vdash b_j \rightsquigarrow v \bullet \sigma''\end{array}}{\sigma \bullet S \vdash a.l_j \rightsquigarrow v \bullet \sigma''} \quad (Red{-}Sel)$$

$$\frac{\sigma \bullet S \vdash a \rightsquigarrow [l_i = \iota_i]^{i\in I} \bullet \sigma' \quad \iota_j \in \mathsf{Dom}(\sigma') \quad j \in I}{\sigma \bullet S \vdash a.l_j \leftarrow \varsigma(x)b \rightsquigarrow [l_i = \iota_i]^{i\in I} \bullet (\sigma'.\iota_j \leftarrow \langle \varsigma(x)b, S\rangle)} \quad (Red{-}Upd)$$

$$\frac{\sigma \bullet S \vdash a \rightsquigarrow [l_i = \iota_i]^{i\in I} \bullet \sigma' \quad \iota_i \in \mathsf{Dom}(\sigma') \quad \iota'_i \notin \mathsf{Dom}(\sigma') \quad \forall i \in I}{\sigma \bullet S \vdash clone(a) \rightsquigarrow [l_i = \iota'_i]^{i\in I} \bullet (\sigma', \iota'_i \mapsto \sigma'(\iota_i))^{i\in I}} \quad (Red{-}Clone)$$

$$\frac{\sigma \bullet S \vdash a \rightsquigarrow v' \bullet \sigma' \quad \sigma' \bullet (S, x \mapsto v') \vdash b \rightsquigarrow v'' \bullet \sigma''}{\sigma \bullet S \vdash let\ x = a\ in\ b \rightsquigarrow v'' \bullet \sigma''} \quad (Red{-}Let)$$

Figure 2: Natural Semantics (NS) of impς

$$\frac{}{\emptyset \vdash \diamond} \quad (Env{-}\emptyset) \qquad \frac{E \vdash A \quad x \notin \mathsf{Dom}(E)}{E, x{:}A \vdash \diamond} \quad (Env{-}Var)$$

$$\frac{E \vdash A_i \quad \forall i \in I}{E \vdash [l_i : A_i]^{i\in I}} \quad (Type{-}Obj) \qquad \frac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C} \quad (Sub{-}Trans)$$

$$\frac{E \vdash A}{E \vdash A <: A} \quad (Sub{-}Refl) \qquad \frac{E \vdash A_i \quad \forall i \in I \cup J}{E \vdash [l_i : A_i]^{i\in I\cup J} <: [l_i : A_i]^{i\in I}} \quad (Sub{-}Obj)$$

Figure 3: Auxiliary Typing Rules

1.2 Type System

The type system for impς is a first-order type system with subtyping. The only type constructor is the one for object types, *i.e.*: $A, B \ ::= \ [l_i : A_i]^{i\in I}$. Notice that the only ground type is [ ]: it can be used as a starting point for building object-types. Other ground types, as `bool`, `int`, `nat`, `real`, ... can can be added at will. The formal typing system is given by the following four judgments: $E \vdash \diamond$, and $E \vdash A$, and $E \vdash A <: B$, and $E \vdash a : A$. The typing environment E consists of a list of assumptions for variables, each of the form $x{:}A$. The typing judgments related to those three judgments are collected in Figure 3. In a nutshell:

- The first judgment $E \vdash \diamond$ describes how to build a *well-formed type environment* E: The empty environment $\emptyset$ is well-formed; a new type binding $x{:}A$ extends a well-formed environment provided that x is fresh and A is a well-formed type.
- The second judgment $E \vdash A$ states that A is a *well-formed type* in the environment E. An object type $[l_i : A_i]^{i\in I}$ is well-formed in the environment E provided that each A_i is well formed in E and that the labels l_i are distinct.
- The third judgment introduces the notion of *subsumption*, which is induced by a *subtype* relation $A <: B$ between object types. An object belonging to a given object type also

$$\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B} \ (Val{-}Sub) \qquad \frac{E', x{:}A, E'' \vdash \diamond}{E', x{:}A, E'' \vdash x : A} \ (Val{-}Var)$$

$$\frac{E, x_i{:}[l_i : B_i]^{i \in I} \vdash b_i : B_i \quad \forall i \in I}{E \vdash [l_i = \varsigma(x_i)b_i]^{i \in I} : [l_i : B_i]^{i \in I}} \ (Val{-}Obj) \qquad \frac{E \vdash a : [l_i : B_i]^{i \in I} \quad j \in I}{E \vdash a.l_j : B_j} \ (Val{-}Sel)$$

$$\frac{E \vdash a : [l_i : B_i]^{i \in I}}{E \vdash clone(a) : [l_i : B_i]^{i \in I}} \ (Val{-}Clone) \qquad \frac{E \vdash a : A \quad E, x{:}A \vdash b : B}{E \vdash let\ x = a\ in\ b : B} \ (Val{-}Let)$$

$$\frac{E \vdash a : [l_i : B_i]^{i \in I} \quad E, x{:}[l_i : B_i]^{i \in I} \vdash b : B_j \quad j \in I}{E \vdash a.l_j \leftarrow \varsigma(x)b : [l_i : B_i]^{i \in I}} \ (Val{-}Upd)$$

Figure 4: The Type Checker for impς

belongs to any supertype of that type and can subsume objects in the supertype, because these have a more limited protocol. The first two rules are the basic rules of reflexivity and transitivity. The third one allows a longer object type to be a subtype of a shorter one. Notice that object types are *invariant* in their component types: the subtyping $[l_i : A_i]^{i \in I \cup J} <: [l_i : B_i]^{i \in I}$ requires $A_i \equiv B_i$ for all $i \in I$. That is, object types are neither covariant neither contravariant in their component types; this condition is necessary in order to guarantee the soundness of the type discipline.

Object typing. The main *term typing judgment* $E \vdash a : A$ states that a has type A in E: the typed rules are presented in Figure 4.

- The rule $(Val{-}Sub)$ connects the typing to the subtyping: an object can emulate another object that has fewer methods, i.e. a more limited protocol;
- The rule $(Val{-}Var)$ is used to extract an assumption from an environment, where x occurs somewhere in;
- According to $(Val{-}Obj)$, an object type $[l_i : B_i]^{i \in I}$ can be assigned to a collection of n methods whose bodies have types $B_1, \ldots, B_n$. Note the "circularity" introduced by the self parameter: in order to give a value a type $[l_i : B_i]^{i \in I}$ the existence of a value of the same type is assumed;
- The rule $(Val{-}Sel)$ tells that, when a method l_j of an object type $[l_i : B_i]^{i \in I}$ is invoked, it produces the corresponding result type B_j;
- Method update $(Val{-}Upd)$ preserves the type of the object that is updated: the type of the object cannot be allowed to change, because other methods assume it;
- The last two rules $(Val{-}Clone)$ and $(Val{-}Let)$ are obvious.

Result and store typing. The typing of results is delicate, because results are pointers to the store, and stores may contain loops. Thus is not possible to determine the type of a result examining its substructures recursively.

The *store types* allow to type the results independently of particular stores: this is possible because type-sound computations do not store results of different types in the same location. A store type associates a *method type* to each store location. Method types have the form $[l_i = B_i]^{i \in I} \Rightarrow B_j$, where $[l_i = B_i]^{i \in I}$ is the type of self and B_j is the result type. It is useful to

$$\frac{M_i \models \diamond \quad \forall i \in I}{\iota_i \mapsto M_i^{i\in I} \models \diamond} \ (Store\text{–}Type) \qquad \frac{j \in I}{[l_i : B_i]^{i\in I} \Rightarrow B_j \models \diamond} \ (Meth\text{–}Type)$$

$$\frac{\Sigma \models \diamond \quad \Sigma_1(\iota_i) = [l_i : \Sigma_2(\iota_i)]^{i\in I} \quad \forall i \in I}{\Sigma \models [l_i = \iota_i]^{i\in I} : [l_i : \Sigma_2(\iota_i)]^{i\in I}} \ (Res\text{–}Obj)$$

$$\frac{\Sigma \models S_i : E_i \quad E_i, x_i{:}\Sigma_1(\iota_i) \vdash b_i : \Sigma_2(\iota_i) \quad \forall i \in I}{\Sigma \models \iota_i \mapsto \langle \varsigma(x_i) b_i, S_i \rangle^{i\in I}} \ (Store\text{–}Typing)$$

$$\frac{\Sigma \models \diamond}{\Sigma \models \emptyset : \emptyset} \ (Stack\text{–}\emptyset\text{–}Typ) \qquad \frac{\Sigma \models S : E \quad \Sigma \models v : A \quad x \notin \mathsf{Dom}(E)}{\Sigma \models S, x \mapsto v : E, x{:}A} \ (Stack\text{–}Var\text{–}Typ)$$

Figure 5: Rules for Store Typing.

introduce also the projections for method types. All type structures are the following:

Σ	::=	$\iota_i \mapsto M_i{}^{i\in I}$	store type	$\Sigma_1(\iota)$	$\triangleq$	$[l_i : B_i]^{i\in I}$ if $\Sigma(\iota) = [l_i : B_i]^{i\in I} \Rightarrow B_j$
M	::=	$[l_i : B_i]^{i\in I} \Rightarrow B_j$	method type	$\Sigma_2(\iota)$	$\triangleq$	B_j if $\Sigma(\iota) = [l_i : B_i]^{i\in I} \Rightarrow B_j$

The formal typing store system is given by the following five judgments: $\Sigma \models \diamond$, and $M \models \diamond$, and $E \models v : A$, and $\Sigma \models \sigma$, and $\Sigma \models S : E$, collected in Figure 5. The *result typing* judgment $\Sigma \models v : A$ means that v has type A with respect to the store type Σ, in the sense that the locations contained in v are assigned types in Σ. Since results are interpreted in stores, it is necessary to capture the compatibility between the store and the store types. This is accomplished by the *store typing* judgment $\Sigma \models \sigma$, whose intended meaning is to check that the contents of every store location has the type of the store type for that location: that is, the method type of the corresponding method closure. The store typing judgment permits the typing of each closure with respect to the whole store, thus accounting for cycles in store. The method body of a closure is typed using a type environment compatible with the stack contained in that closure. This compatibility is described by the *stack typing* judgment $\Sigma \models S : E$, which is defined via the result typing.

Subject reduction. The Type Soundness of the typing discipline is an ultimate metatheoretical property: the successful static typing of programs is a proof of partial correctness about their execution. The Type Soundness for the $\mathsf{imp}\varsigma$-calculus ensures that every well-typed and not diverging term never yields the *message-not-found* runtime error. This is an immediate consequence of the subject reduction theorem; the statement and the proof of the subject reduction require a preliminary definition and lemma all proved in [1].

Definition 1 (Store type extension) *We say that $\Sigma' \geq \Sigma$ (Σ' is an extension of Σ) if and only if $\mathsf{Dom}(\Sigma) \subseteq \mathsf{Dom}(\Sigma')$ and for all $\iota \in \mathsf{Dom}(\Sigma)$: $\Sigma'(\iota) = \Sigma(\iota)$.*

Lemma 1 (Stack typing, Bound weakening) *1. If $\Sigma \models S : E$ and $\Sigma' \models \diamond$, with $\Sigma' \geq \Sigma$, then $\Sigma' \models S : E$;*

2. If $E, x : D, E' \vdash \mathcal{I}$ and $E \vdash D' <: D$, then $E, x : D', E' \vdash \mathcal{I}$.

The Subject Reduction states that the dynamic semantics is consistent with the type system.

Theorem 1 (Subject Reduction) *If $E \vdash a : A$, and $\sigma \bullet S \vdash a \rightsquigarrow v \bullet \sigma'$, and $\Sigma \models \sigma$, and $\mathsf{Dom}(\sigma) = \mathsf{Dom}(\Sigma)$, and $\Sigma \models S : E$, then there exist a type A', and a store type Σ', such that $\Sigma' \geq \Sigma$, and $\Sigma' \models \sigma'$, $\mathsf{Dom}(\sigma') = \mathsf{Dom}(\Sigma')$, and $\Sigma' \models v : A'$, and $A' <: A$.*

It is immediate to deduce that if a closed term has a type and the term reduces to a result in a store, then the result can be assigned that type in that store. Equivalently, if a closed term produces a result, it does so respecting the type that it had been assigned statically.

Corollary 1 (Subject Reduction for closed terms) *If $\emptyset \vdash a : A$, and $\emptyset \bullet \emptyset \vdash a \rightsquigarrow v \bullet \sigma$, then there exist a type A' and a store type Σ' such that $\Sigma' \models \sigma$, and $\Sigma' \models v : A'$, with $A' <: A$*

The statement of the previous corollary is vacuous if the starting term does not produce a result: this can happen either because the reduction diverges —the rules are applicable ad infinitum— or because it gets stuck —no rule is applicable at a certain stage. But the latter case is not possible, which is the essence of the Type Soundness property.

Theorem 2 (Type Soundness) *The reduction of a not diverging well-typed term of impς in a well-typed store cannot get stuck, and produces a result of the expected type.*

2 impς^{nov} in Coinductive Natural Deduction Semantics

In order to make the formalization in Coq easier, we reformulate the semantics and typing systems of impς in a form that takes full advantage of all proof-theoretical concepts provided by modern coinductive type theories (such as $\text{CC}^{(\text{Co})\text{Ind}}$), namely natural deduction style, higher-order abstract syntax and coinductive types. This setting, which we call *coinductive natural deduction semantics*, is particularly successful for the fragment of impς without object override, denoted by impς^{nov}. In this case, the resulting system is very clean and compact, allowing for an easier treatment of theoretical and metatheoretical results.

The key point in using the Natural Deduction Semantics (NDS) style [7,25] is that all stack-like structures (*e.g.*, environments) are distributed in the hypotheses of proof derivations. Therefore, judgments, rules and proofs we have to deal with are much simpler. On the other hand, we have to address some consequences of using a distributed setting, instead of carrying locally all the information we need. The major changes concern the operational semantics, whereas the type system for terms needs a simpler reformulation. Finally, very delicate is the typing of results, which makes use also of coinductive proof techniques.

Syntax. Following the higher-order abstract syntax paradigm [17,28], we reduce all binders to the sole λ-abstraction. Therefore, from now on we write $let(a, \lambda x.b)$ for $let\ x = a\ in\ b$, and $\varsigma(x)b$ for $\varsigma(\lambda x.b)$, where $let : Term \times (Var \to Term) \to Term$, and $\varsigma : (Var \to Term) \to Term$. (We will keep using the notation "$\varsigma(x)b$" as syntactic sugar). Of course, the usual conventions about α-conversion apply.

2.1 Natural Deduction Semantics

The *term reduction* judgment of impς $\sigma \bullet S \vdash a \rightsquigarrow v \bullet \sigma'$ is translated as $\Gamma \vdash eval(s, a, s', v)$; that is, we model the operational semantics by a predicate *eval* defined on 4-tuples $eval \subseteq Store \times Term \times Store \times Res$. Γ is the *proof derivation contexts*, that is a set of assertions (of any judgment) which can be used as assumptions in the proof derivations. The intended meaning of the derivation $\Gamma \vdash eval(s, a, s', v)$ is that, starting with the store s and using the assumptions in Γ, the term a reduces to a result v, yielding an updated store s'. The rules for *eval* are in Figure 6. As usual in Natural Deduction, rules are written in "vertical" notation, *i.e.*, the hypotheses of a derivation $\Gamma \vdash \mathcal{J}$ are distributed on the leaves of the proof tree.

Notice that the stack S disappears from the judgment *eval*. Its content is distributed in Γ, *i.e.* Γ contains enough assumptions to carry the association between variables and values. These bindings can be created in the form of hypothetical premises local to sub-reductions, discharged in the spirit of natural deduction style—see *e.g.* rules (*e_let*) and (*e_clone*). It is worth noticing that we do not need to introduce the well-formedness judgments for stores and environments: these properties will be automatically ensured by the freshness conditions of *eigenvariables* in the natural deduction style.

A consequence of NDS is that closures cannot be a pair "method, stack", because there are no explicit stacks to put in anymore. Rather, we have to "calculate" closures by gathering from the environment the values associated to free variables of methods bodies. Thus closures are translated as $\langle \varsigma(x)b, S \rangle \longmapsto \lambda x.bind(v_1, (\lambda y_1.bind(\ldots bind(v_n, (\lambda y_n.ground(b)))\ldots)))$, where the first (*i.e.* outer) abstraction λx stands for $\varsigma(x)$, and the n remaining abstractions ($n \geq 0$) capture

$$\frac{x \mapsto v}{eval(s, x, s, v)}\ (e\_var) \qquad \frac{\begin{array}{c}(closed(x_i))\\ \vdots \\ \iota_i \notin \mathsf{Dom}(s) \quad wrap(b_i, \overline{b}_i) \quad \forall i \in I\end{array}}{eval(s, [l_i = \varsigma(x_i)b_i]^{i\in I}, (s, \iota_i \mapsto \lambda x_i.\overline{b}_i), [l_i = \iota_i]^{i\in I})}\ (e\_obj)$$

$$\frac{\begin{array}{c}(x \mapsto v)\\ \vdots \\ eval(s', b(x), s'', v') \\ eval(s, a, s', v) \quad (e\_let)\end{array}}{eval(s, let(a, b), s'', v')}\ (e\_let) \qquad \frac{\begin{array}{c}(x \mapsto [l_i : \iota_i]^{i\in I})\\ \vdots \\ eval_b(s', \overline{b}_j, s'', v) \qquad j \in I \\ eval(s, a, s', [l_i : \iota_i]^{i\in I}) \quad s'(\iota_j) = \lambda x.\overline{b}_j\end{array}}{eval(s, a.l_j, s'', v)}\ (e\_call)$$

$$\frac{\begin{array}{c}\iota'_i \notin \mathsf{Dom}(s') \\ eval(s, a, s', [l_i = \iota_i]^{i\in I}) \quad \iota_i \in \mathsf{Dom}(s') \quad \forall i \in I\end{array}}{eval(s, clone(a), (s', \iota'_i \mapsto s'(\iota_i)), [l_i = \iota'_i]^{i\in I})}\ (e\_clone)$$

$$\frac{eval(s, a, s', v)}{eval_b(s, ground(a), s', v)}\ (e\_ground) \qquad \frac{\begin{array}{c}(x \mapsto v)\\ \vdots \\ eval_b(s, \overline{b}, s', v')\end{array}}{eval_b(s, bind(v, \lambda x.\overline{b}), s', v')}\ (e\_bind)$$

$$\frac{\begin{array}{c}(closed(x_i))\\ \vdots \\ closed(b_i) \quad \forall i \in I\end{array}}{closed([l_i = \varsigma(x_i)b_i]^{i\in I})}\ (c\_obj) \qquad \frac{\begin{array}{c}(closed(x))\\ \vdots \\ closed(a) \quad closed(b(x))\end{array}}{closed(let(a, b))}\ (c\_let)$$

$$\frac{closed(a)}{closed(a.l)}\ (c\_call) \qquad \frac{closed(a)}{closed(clone(a))}\ (c\_clone)$$

$$\frac{closed(b)}{wrap(b, ground(b))}\ (w\_ground) \qquad \frac{\begin{array}{c}(closed(z))\\ \vdots \\ wrap(b\{z/y\}, \overline{b}\{z/y\}) \quad y \mapsto v \quad z\ fresh\end{array}}{wrap(b, bind(v, \lambda y.\overline{b}))}\ (w\_bind)$$

Figure 6: Natural Deduction Semantics (NDS) of $\mathsf{imp}\varsigma^{\mathrm{nov}}$

the free variables of b. Hence, *bind* and *ground* are the two constructors of a new syntactic sort, *i.e.* $\overline{b} ::= \lambda x.ground(b) \mid \lambda x.bind(v, \overline{b})$. Closures are dealt with by two auxiliary judgments *wrap* and $eval_b$, *i.e.* $wrap \subseteq Term \times Body$ and $eval_b \subseteq Store \times Body \times Store \times Res$. The judgment *wrap* implements the formation of closure *bodies*, that is terms where the only possibly free variable is the one corresponding to *self*. The intended meaning of $\Gamma \vdash wrap(b, c)$ is that c is a closure body obtained by binding all free variables in the term b to their respective results in Γ. We need therefore to keep track of free variables in terms; to this end, we introduce a judgment $closed \subseteq Term$, whose formal meaning is $\Gamma \vdash closed(t) \iff$ for all $x \in \mathsf{FV}(t) : closed(x) \in \Gamma$. Intuitively, the rules of *wrap* allow for successively binding the free variables appearing in the method body, until it is "closed". When we apply rule (w_bind), we choose any (free) variable y in b, and bind it to the corresponding value v, as stated in Γ. The remaining part of the closure $\overline{b}$ can be seen as the closure body of a method where the variable y is supposed to be "closed", and therefore it is obtained in a proof environment containing this information. This is obtained by the subderivation $\Gamma, closed(z) \vdash wrap(b\{z/y\}, \overline{b}\{z/y\})$, which repeats the rule (w_bind) until enough variables have been bound and, correspondingly, enough assumptions of the form $closed(z)$ have

been taken to be able to prove $closed(b)$ (*i.e.* there are no more free variables to bind) and thus apply rule (w_ground).

Notice that the closures we get in this manner are "optimized", because only variables which are really free in the body need to be bound in the closure, although in a non-deterministic order. Evaluation of a closure takes place in the rule of method selection (e_call), in a context extended with the binding between a fresh variable (representing *self*) and the (implementation of the) host object. Of course, all the local bindings of the closure have to be unraveled (*i.e.* assumed in the hypotheses) before the real evaluation of the body is performed. This unraveling is implemented by the auxiliary judgment $eval_b$, which can be seen as the dual of $wrap$.

For lack of space, we cannot describe in detail all the rules of Figure 6; we refer to [9] for a complete discussion. We limit ourselves to the object creation rule (e_obj). The semantics of objects requires the preliminary transformation of the method list, forming the object, into a closure list: this is obtained through the $wrap$ judgment and corresponding hypothetical premises. Notice that the well-formedness condition about the stack disappears, since they are automatically ensured by the natural deduction style.

Adequacy. We prove here that the presentation of $\mathsf{imp}\varsigma$ in NDS corresponds to the original one of [1]. Due to lack of space, we cannot describe these results in detail; we refer the interested reader to [9]. First, we establish the relationship between our heterogeneous contexts Γ and the environments S, E of the original setting [1], and between the two kinds of stores s and σ.

Definition 2 (Well-formed context) *A context Γ is well-formed if it can be partitioned as $\Gamma = \Gamma_{Res} \cup \Gamma_{TType} \cup \Gamma_{closed}$, where Γ_{Res} contains only formulae of the form $x \mapsto v$, and Γ_{TType} contains only formulae of the form $x \mapsto A$, and Γ_{closed} contains only formulae of the form $closed(x)$; moreover, Γ_{Res} and Γ_{TType} are functional (e.g., if $x \mapsto v, x \mapsto v' \in \Gamma_{Res}$ then $v \equiv v'$).*

Definition 3 *Let Γ be a context, S a stack, E a type environment, s and σ stores. We define the following:*

$$\Gamma \subseteq S \triangleq \forall x \mapsto v \in \Gamma.\ x \mapsto v \in S \qquad \Gamma \subseteq E \triangleq \forall x \mapsto A \in \Gamma.\ x \mapsto A \in E$$

$$S \subseteq \Gamma \triangleq \forall x \mapsto v \in S.\ x \mapsto v \in \Gamma \qquad E \subseteq \Gamma \triangleq \forall x \mapsto A \in E.\ x \mapsto A \in \Gamma$$

$$\gamma(S) \triangleq \{x \mapsto S(x) \mid x \in \mathsf{Dom}(S)\}$$

$$s \lesssim \sigma \triangleq \forall \iota_i \in \mathsf{Dom}(s).\ \gamma(S_i), closed(x) \vdash wrap(b_i, s(\iota_i)(x_i)), \qquad where\ \sigma(\iota_i) = \langle \varsigma(x_i) b_i, S_i \rangle$$

$$\sigma \lesssim s \triangleq \forall \iota_i \in \mathsf{Dom}(\sigma).\ \gamma(S_i), closed(x) \vdash wrap(b_i, s(\iota_i)(x_i)), \qquad where\ \sigma(\iota_i) = \langle \varsigma(x_i) b_i, S_i \rangle$$

In the following theorem, for $\bar{b}$ a closure, let us denote by $stck(\bar{b})$ the stack containing the bindings in $\bar{b}$, and by $body(\bar{b})$ the inner body. These functions can be defined recursively on $\bar{b}$ as follows:

$$stck(ground(b)) = \emptyset \qquad stck(bind(v, \lambda x.\bar{b})) = stck(\bar{b}) \cup \{x \mapsto v\}$$
$$body(ground(b)) = b \qquad body(bind(v, \lambda x.\bar{b})) = body(\bar{b})$$

Theorem 3 (Adequacy of reduction) *Let Γ be well-formed, and $\sigma \bullet S \vdash \diamond$.*

1. *Let $\Gamma \subseteq S$, and $s \lesssim \sigma$.*
   (a) *If $\Gamma \vdash eval(s, a, s', v)$, then there exists σ' such that $\sigma \bullet S \vdash a \rightsquigarrow v \bullet \sigma'$, and $s' \lesssim \sigma'$;*
   (b) *If $\Gamma \vdash eval_b(s, \bar{b}, s', v)$, then there exists σ', such that $\sigma \bullet stck(\bar{b}) \vdash body(\bar{b}) \rightsquigarrow v \bullet \sigma'$, and $s' \lesssim \sigma'$.*
2. *Let $S \subseteq \Gamma$, and $\sigma \lesssim s$.*
   If $\sigma \bullet S \vdash a \rightsquigarrow v \bullet \sigma'$, then there exists s', such that $\Gamma \vdash eval(s, a, s', v)$, and $\sigma' \lesssim s'$.

Proof 1 1. *By mutual structural induction on $\Gamma \vdash eval(s, a, s', v)$ and $\Gamma \vdash eval_b(s, \bar{b}, s', v)$;*

2. *By structural induction on $\sigma \bullet S \vdash a \rightsquigarrow v \bullet \sigma'$.* □

$$\frac{wt(B_i) \quad \forall i \in I}{wt([l_i : B_i]^{i\in I})} \ (wt\_obj) \qquad \frac{sub(A,B) \quad sub(B,C)}{sub(A,C)} \ (sub\_trans)$$

$$\frac{wt(B_i) \quad \forall i \in I \cup J}{sub([l_i : B_i]^{i\in I\cup J}, [l_i : B_i]^{i\in I})} \ (sub\_obj) \qquad \frac{wt(A)}{sub(A,A)} \ (sub\_refl)$$

$$\frac{type(a,A) \quad sub(A,B)}{type(a,B)} \ (t\_sub) \qquad \frac{type(a,[l_i : B_i]^{i\in I}) \quad j \in I}{type(a.l_j, B_j)} \ (t\_call)$$

$$\frac{wt(A) \quad x \mapsto A}{type(x,A)} \ (t\_var) \qquad \frac{type(a,[l_i : B_i]^{i\in I})}{type(clone(a),[l_i : B_i]^{i\in I})} \ (t\_clone)$$

$$\frac{\begin{array}{c}(x_i \mapsto [l_i : B_i]^{i\in I}) \\ \vdots \\ type(b_i, B_i) \qquad \forall i \in I\end{array}}{type([l_i = \varsigma(x_i)b_i]^{i\in I}, [l_i : B_i]^{i\in I})} \ (t\_obj) \qquad \frac{\begin{array}{c}(x \mapsto A) \\ \vdots \\ type(a,A) \quad type((b\ x), B)\end{array}}{type(let(a,b), B)} \ (t\_let)$$

Figure 7: Term Typing for $\mathsf{imp}\varsigma^{\mathrm{nov}}$

2.2 Typing of Terms

The original type system for terms is easily translated in NDS. The *term typing* judgment $E \vdash a : A$ is transformed as $\Gamma \vdash type(a, A)$, where $type \subseteq Term \times TType$ and $TType$ is the sort of (term) types. Also the other judgments of *well-formedness* of types $wt \subseteq TType$ and of *subtype* $sub \subseteq TType \times TType$ are recovered also in this setting.

As the stack S disappears from the reduction judgment, so the type environment E disappears from the typing judgment, thus simplifying the judgment itself and the formal proofs about it. The global context Γ contains, among other stuff (knowledge, information), associations between the free variables x_i, eventually appearing in the object a, and the corresponding types A_i. These typing assignments will be used, locally to sub-reductions, for assuming hypothetical premises about the types of variables, and then will be discharged according to the Natural Deduction style. dunque non giudizio (derivato, come per S) of stacks, we do not need to introduce a well-formedness judgment for the type environments. Typing contexts will be automatically ensured to be well-formed by the stack discipline of Natural Deduction and freshness of locally-quantified variables.

The rules in NDS for typing terms and related judgments are given in Figure 7. They are an easy translation of the original ones; just notice that in rules (t_obj) and (t_let) we discharge a typing assumption on a locally-quantified (*i.e.*, fresh) variable. Moreover, in rule (t_var), the premise $wt(A)$ ensures that only well-formed types can be used for typing terms.

Theorem 4 (Adequacy of term typing) *Let Γ be well-formed, and E such that $E \vdash \diamond$.*

1. *If $\Gamma \subseteq E$, and $\Gamma \vdash type(a, A)$, then $E \vdash a : A$;*
2. *If $E \subseteq \Gamma$, and $E \vdash a : A$, then $\Gamma \vdash type(a, A)$.*

Proof 2 1. *By structural induction on the derivation of $\Gamma \vdash type(a, A)$;*

2. *By structural induction on the derivation of $E \vdash a : A$.* □

$$\frac{\begin{array}{llll} v \equiv [l_i = \iota_i]^{i\in I} & A \equiv [l_i : B_i]^{i\in I} & \begin{array}{c}(x_i \mapsto A), (cores(s, v, A))\\ \vdots\end{array} & \\ s(\iota_i) \equiv \lambda x_i.\overline{b}_i & wt([l_i : B_i]^{i\in I}) & cotype_b(s, b_i, B_i) & \iota_i \in \mathsf{Dom}(s) \quad \forall i \in I \end{array}}{cores(s, v, A)} \; (t\_cores)$$

$$\frac{type(b, A)}{cotype_b(s, ground(b), A)} \; (t\_coground) \qquad \frac{\begin{array}{cc} & (y \mapsto A)\\ & \vdots\\ cores(s, v, A) & cotype_b(s, \overline{b}, B)\end{array}}{cotype_b(s, bind(v, \lambda y.\overline{b}), B)} \; (t\_cobind)$$

Figure 8: Coinductive Rules for *cores* and $cotype_b$.

2.3 Coinductive Typing of Results

Since results contain references to the store, in order to be able to type them we need to type store locations. In principle, the type of a store location is the type of its content, which in turn may contain pointers to the store. Therefore, the potential presence of loops in the store makes the typing system for results non-trivial: a naïve system would chase pointers indefinitely, unraveling non-wellfounded structures in the memory. The solution adopted in [1] is to introduce yet another typing structure, the *store types*, which assign to each location a type consistently with the content of the location. Store types have to be provided beforehand, and suitable auxiliary proof systems are needed in order to check their consistency with the stores. Of course all this technical machinery adds further complexity to the (already difficult, though) metatheoretical properties of the system.

In this paper, we propose a different approach to result typing inspired by the features of $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, where the canonical way for dealing with non-wellfounded, circular data is *coinduction*. We will see now that this approach is quite successful for the fragment $\mathsf{imp}\varsigma^{\mathrm{nov}}$, although is not sufficient for the full $\mathsf{imp}\varsigma$.

For $\mathsf{imp}\varsigma^{\mathrm{nov}}$, the two original judgments of *result typing* $\Sigma \models v : A$ and *store typing* $\Sigma \models s$ collapse into a unique judgment: $\Gamma \vdash cores(s, v, A)$. More precisely we use two mutual coinductive judgments $cores \subseteq Store \times Res \times TType$, and $cotype_b \subseteq Store \times Body \times TType$. The intended meaning of the derivation of $\Gamma \vdash cores(s, v, A)$ is that the object v in the store s has type A; similarly, $\Gamma \vdash cotype_b(s, b, A)$ means that the body of some object's method b in the store s as type A. The rules for these mutual coinductive judgments are in Figure 8. The idea at the heart of this system is simple and it can be caught by looking at rule (t_cores): in order to check if an object can be given a type A, we open all the pointers belonging to method closures, thus visiting the store until a closure without pointers is reached: then we can type a closure using the traditional *type* judgment. If the original pointer is encountered in the meanwhile, then the type A we started with is used (see rule (t_cobind)). Clearly, this means that the predicate we are proving has to be *assumed* in the hypotheses, hence the coinduction.

As a result, the typing system is very simple, and moreover we do not need store types (and all related machinery) anymore. Also, since stacks and type environments are already distributed in the proof contexts, we do not need *stack typing* judgments either; however, in stating and proving the Subject Reduction theorem we will require that the types of variables in the contexts will be consistent with the values associated to variables.

Let us see an example of application of the coinductive rules. We type the result $[l = 0]$ to the store (containing a loop) $s \equiv \{0 \mapsto \lambda x.bind([l = 0], \lambda y.ground(y))\}$ with the following derivation:

$$\dfrac{\dfrac{(cores(s,[l=0],[l:[\,]]))_{(1)}}{\dfrac{cotype_b(s, bind([l=0], \lambda y.ground(y))\ [\,])}{cores(s,[l=0],[l:[\,]])}\ (t\_cores)\ (1)}\ (t\_cobind)\ (2)}{\dfrac{\dfrac{(y \mapsto [l:[\,]])_{(2)}}{type(y,[l:[\,]])}\ (t\_var) \qquad sub([l:[\,]],[\,])}{type(y,[\,])}\ (t\_sub)}{cotype_{body}(s, ground(y), [\,])}\ (t\_coground)$$

Adequacy. Since we use coinductive proof systems, our perspective is quite different from the original formulation of impς. Nevertheless, for the $\text{imp}\varsigma^{\text{nov}}$ fragment we have the following adequacy result:

Theorem 5 (Adequacy of coinductive result typing) *Let Γ be a well-formed context. Then:*

1. *For $s \lesssim \sigma$, if $\Gamma \vdash cores(s, v, A)$, then there exists Σ, such that $\Sigma \models v : A$, and $\Sigma \models \sigma$;*
2. *For $\sigma \lesssim s$, if $\Sigma \models v : A$, and $\Sigma \models \sigma$, then $\Gamma \vdash cores(s, v, A)$.*

Proof 3 *(1) By induction on v; (2) By induction on the structure of the derivation $\Sigma \models v : A$.* □

2.4 Subject Reduction

We can now state and prove the Subject Reduction. Due to the rephrasing of impς, we have to require the coherence between the types and the results associated to variables in the context; that is, given the store s:

$$\forall x, w, C.\ x \mapsto w, x \mapsto C \in \Gamma \Rightarrow \Gamma \vdash cores(s, w, C)$$

This corresponds to the (Store Typing) judgment of [1], but our management, thanks to distributed stacks and environments, is easier. We obtain the following version of the Subject Reduction theorem, which is simpler both to state and prove:

Theorem 6 (Subject Reduction for $\text{imp}\varsigma^{\text{nov}}$) *Let Γ be a well-formed context. Then:*

$$\begin{array}{l}\Gamma \vdash type(a, A)\ \wedge\ \Gamma \vdash eval(s, a, t, v)\ \wedge \\ (\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash cores(s, w, C))\ \Rightarrow \\ \exists A^+ : TType.\ \Gamma \vdash cores(t, v, A^+)\ \wedge\ \Gamma \vdash sub(A^+, A).\end{array}$$

Proof 4 *By structural induction on the derivation of $\Gamma \vdash eval(s, a, t, v)$. See Appendix C.* □

3 impς in Inductive Natural Deduction Semantics

In this section, we scale up from $\text{imp}\varsigma^{\text{nov}}$ toward the full impς, taking into account the "override" constructor. This apparently slight difference yields some deep changes in the typing system of results, because we need to introduce new syntactic structures and the result typing system is not coinductive anymore. Consequently, also the properties of the calculus need to be reformulated; in spite of this, most of the development carried out for $\text{imp}\varsigma^{\text{nov}}$ can be readily recovered for the full impς, thus enlightening the modularity of our development.

$$\frac{\begin{array}{c}(closed(x))\\ \vdots\end{array}}{\begin{array}{c}eval(s,a,s',[l_i=\iota_i]^{i\in I}) \quad j\in I \quad \iota_j\in \mathsf{Dom}(s') \quad wrap(b,\overline{b})\\ \hline eval(s,a.l\leftarrow\varsigma(x)b,(s'.\iota_j\leftarrow\lambda x.\overline{b}),[l_i=\iota_i]^{i\in I})\end{array}}\ (e\_over)$$

$$\frac{\begin{array}{c}closed(x)\\ \vdots\\ closed(a)\quad closed(b)\end{array}}{closed(a.l\leftarrow\varsigma(x)b)}\ (c\_over) \qquad \frac{\begin{array}{c}(x\mapsto[l_i:B_i]^{i\in I})\\ \vdots\\ type(a,[l_i:B_i]^{i\in I}) \quad j\in I \quad type(b,B_j)\end{array}}{type(a.l\leftarrow\varsigma(x)b,[l_i:B_i]^{i\in I})}\ (t\_over)$$

Figure 9: Extra rules for operational semantics and typing of terms for $\mathsf{imp}\varsigma$.

$$\frac{\Sigma_1(\iota_i)\equiv[l_i:\Sigma_2(\iota_i)]^{i\in I} \quad wt([l_i:\Sigma_2(\iota_i)]^{i\in I}) \quad \iota_i\in\mathsf{Dom}(\Sigma) \quad \forall i\in I}{res(\Sigma,[l_i=\iota_i]^{i\in I},[l_i:\Sigma_2(\iota_i)]^{i\in I})}\ (t\_res)$$

$$\frac{dom(\Sigma)\subseteq dom(\Sigma') \quad \forall\iota\in dom(\Sigma).\ \Sigma'(\iota)=\Sigma(\iota)}{ext(\Sigma',\Sigma)}\ (t\_ext) \qquad \frac{type(b,A)}{type_b(\Sigma,ground(b),A)}\ (t\_ground)$$

$$\frac{\begin{array}{c}(x_i\mapsto\Sigma_1(\iota_i))\\ \vdots\\ dom(s)\subseteq dom(\Sigma) \quad type_b(\Sigma,s(\iota_i)(x_i),\Sigma_2(\iota_i)) \quad \forall i\in I\end{array}}{comp(\Sigma,s)}\ (t\_comp) \qquad \frac{\begin{array}{c}(y\mapsto A)\\ \vdots\\ res(\Sigma,v,A) \quad type_b(\Sigma,\bar{b},B)\end{array}}{type_b(\Sigma,bind(v,\lambda y.\bar{b}),B)}\ (t\_bind)$$

Figure 10: Inductive rules for result typing for $\mathsf{imp}\varsigma$.

3.1 Natural Deduction Semantics and Typing of Terms

The operational semantics of Figure 6 and the typing system of Figure 7 are immediately extended to $\mathsf{imp}\varsigma$ by adding the rules dealing with override, as shown in Figure 9. The adequacy results for the reduction and term typing judgment of $\mathsf{imp}\varsigma^{\mathrm{nov}}$ (Theorems 3, 4) naturally extends to $\mathsf{imp}\varsigma$.

3.2 Inductive Typing of Results

Let $SType$ be the sort of *stores types*, that is finite maps from locations to method types. The *result typing* judgment is a predicate defined on triples $res\subseteq SType\times Res\times TType$. The intended meaning of a derivation $\Gamma\vdash(res\ \Sigma\ v\ A)$ is that in the store type Σ, the result v has type A. That is, for all $\iota_i\in\pi_2(v)$: $A=\Sigma_1(\iota_i)$. In the (formal) development of the theory of $\mathsf{imp}\varsigma$, we are interested only stores whose content is "compatible" with store types. This compatibility is represented by the judgment $comp\subseteq SType\times Store$, which corresponds to $\Sigma\models s$. If $\Gamma\vdash comp(\Sigma,s)$, then the content of each locations in the store s can be given the type indicated by Σ. Finally, we need the auxiliary *extension* relation $ext\subseteq SType\times SType$ ($\Gamma\vdash ext(\Sigma',\Sigma)$ means that Σ' extends Σ) and a specialized predicate for typing closures, namely $type_b\subseteq SType\times Body\times TType$. The rules for the judgments res, ext, $comp$ and $type_b$ are in Figure 10. Some comments on this system are in order. First, we do not need to formalize the *well-formedness of store types*. As before, this property can be proved to hold automatically, due to the use of natural deduction style. We do not need the "stack typing" judgment either—in fact, we do not have explicit stacks and type environments at all. However, the correspondence between results and types associated to the same variable in the proof environment will be taken into account in the Subject Reduction theorem.

An important difference of this system *w.r.t.* the corresponding one for $\mathsf{imp}\varsigma^{\mathrm{nov}}$ (Figure 8), is that the rules in Figure 10 are in the usual inductive setting. We do not need the coinductive approach, because we do not check the types of locations by chasing pointers in the store; instead,

we resort to the store type Σ, which is a finite structure. Of course, store types have to be given beforehand, that is they cannot be synthesized by the typing system.

We prove the adequacy of the reformulation: notice that the adequacy holds also for the fragment without object override, because the present calculus is a conservative extension of $\mathsf{imp}\varsigma^{\mathrm{nov}}$.

Theorem 7 (Adequacy of inductive result typing) *Let Γ be a well-formed context, and Σ a store type such that $\Sigma \vdash \diamond$.*

1. *For $s \lesssim \sigma$, if $\Gamma \vdash res(\Sigma, v, A)$ and $\Gamma \vdash comp(\Sigma, s)$, then $\Sigma \models v : A$ and $\Sigma \models \sigma$;*
2. *For $\sigma \lesssim s$, if $\Sigma \models v : A$ and $\Sigma \models \sigma$, then $\Gamma \vdash res(\Sigma, v, A)$ and $\Gamma \vdash comp(\Sigma, s)$.*

Proof 5 1. *By induction on the structure of the derivations $\Gamma \vdash res(\Sigma, v, A)$ and $\Gamma \vdash comp(\Sigma, s)$;*

2. *By induction on the structure of the derivations $\Sigma \models v : A$ and $\Sigma \models \sigma$.* □

3.3 Subject Reduction

We can now state and prove the Subject Reduction theorem. Similarly to the case of Theorem 6, we have to take care of the coherence between the types and the values associated to variables in the context: that is, given the store type Σ:

$$\forall x, w, C.\ x \mapsto w, x \mapsto C \in \Gamma \Rightarrow \Gamma \vdash res(\Sigma, w, C)$$

It is worth noticing that the Subject Reduction is more involved than the corresponding one for the coinductive setting (Theorem 6), both in the statement and in the proof, due to the presence of store types.

Theorem 8 (Subject Reduction for $\mathsf{imp}\varsigma$) *Let Γ be a well-formed context. Then:*

$$\begin{array}{l}\Gamma \vdash type(a, A) \ \wedge\ \Gamma \vdash eval(s, a, t, v) \ \wedge\ \ \Gamma \vdash comp(\Sigma, s) \ \wedge \\ (\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash res(\Sigma, w, C)) \ \Rightarrow \\ \exists A^+ : TType, \Sigma^+ : SType. \\ \Gamma \vdash res(\Sigma^+, v, A^+) \ \wedge\ \Gamma \vdash ext(\Sigma^+, \Sigma) \ \wedge\ \Gamma \vdash comp(\Sigma^+, t) \ \wedge\ \Gamma \vdash sub(A^+, A)\end{array}$$

Proof 6 *By structural induction on the derivation of $\Gamma \vdash eval(s, a, t, v)$. See Appendix D.* □

4 Formalization in Coq

In the previous sections we have given a presentation of $\mathsf{imp}\varsigma$ better suited for an encoding in Coq. Its formalization in the specification language of the proof assistant is nevertheless a complex task, since we have to face many subtle details which are left "implicit" on the paper. Due to lack of space, here we will briefly discuss only few interesting features of this development; we refer to [9] for further details.

The encoding of $\mathsf{imp}\varsigma$ in Coq follows naturally the rephrasing of the previous sections. An immediate advantage is that we can use well-known encoding methodologies (see *e.g.* [17,26,30,28], among others); therefore, the adequacy of the encoding with respect to the NDS presentation can be proved easily.

A well-known problem we have to face in the encoding is the treatment of the binders, namely ς and *let*. Binders are known to be difficult to deal with; we would the metalanguage to take care of all the burden of α-conversion, substitutions, variable scope and so on. In recent years, many approaches have been proposed for dealing with binders, which essentially differ from the expressive power of the underlying metalanguage; a detailed description of these approaches is outside the scope of this paper. We refer to e.g. [18, 17, 26, 27, 28, 19, 20] for a deeper discussion.

Among the many possibilities, we have chosen the *second-order abstract syntax*, called also "weak HOAS" [26, 20, 19]. In this approach, binding operators are represented by constructors of higher order type [12, 26]. The main difference with respect to the full HOAS is that abstractions range over unstructured (*i.e.*, non inductive) sets of *abstract variables*. In this way, α-conversion is automatically provided by the metalanguage, while substitution of terms for variables is not. This fits perfectly the needs for the encoding of impς, since the language is taken up-to α-equivalence, and substitution is never used in the semantics. Moreover the weak HOAS is compatible with inductive datatypes, still avoiding the arising of *exotic terms*; therefore, we can establish easily the adequacy of the encoding.

An issue of HOAS is related to the difficulty of reasoning by induction and using recursion over contexts, since they are rendered as functional terms. Finally, one looses the possibility of handling and proving properties over the mechanisms delegated to the metalanguage.

However, when we have to develop metatheoretical results in a HOAS setting, the expressive power of CC$^{(Co)Ind}$ may be not enough. In [20, 19, 30] a general methodology for reasoning on systems in HOAS is presented. The gist is to extend the framework with a set of axioms, called the *Theory of Contexts*, capturing some basic and natural properties of *names* and *term contexts*. These axioms allow for a smooth handling of schemata in HOAS, with a very low mathematical and logical overhead. In fact, this work can be seen also as an extensive case study in the application of the Theory of Contexts, applied here for the first time on an imperative object-oriented calculus.

4.1 Syntax

The signature of the weak HOAS-based encoding of the syntax is the following:

```
Parameter  Var : Set.
Definition Lab := nat.
Inductive Term : Set := var  : Var -> Term
                      | obj  : Object -> Term
                      | call : Term -> Lab -> Term
                      | over : Term -> Lab -> (Var -> Term) -> Term
                      | clone: Term -> Term
                      | let  : Term -> (Var -> Term) -> Term
with Object : Set := obj_nil : Object
                   | obj_cons: Lab -> (Var -> Term) -> Object -> Object.
Coercion var : Var >-> Term.

Inductive TType : Set := mktype: (list (Lab * TType)) -> TType.
```

Notice first that we use a separate type Var for variables (weak HOAS setting): the only terms which can inhabit this type are the variables of the metalanguage, that represent directly the variables of the object language. The α-equivalence on terms is immediately inherited from the metalanguage, still keeping induction principles for term. For instance, the two objects $[m = \varsigma(x)x]$ and $[m = \varsigma(y)y]$ can be represented by: `(obj (obj_cons m [x:Var]x obj_nil))`, and `(obj (obj_cons m [y:Var]y obj_nil))` which are α-equivalent.

Notice that, if Var were inductive, we could define *exotic terms* using the Case construct of Coq. Exotic terms are Coq terms which do not correspond to any expression of impς, and therefore they have to be ruled out by extra "well-formedness" judgments, thus complicating the whole encoding.

4.2 Natural Deduction Semantics

In order to encode the operational semantics of impς, we have to represent all the required entities and operations for their manipulation. Due to lack of space, we present here only a selection of these datatypes.

Locations are naturally represented as natural numbers. Results are lists of pairs built of method names and pointers to the corresponding closures in the store:

```
Definition Loc   := nat.
Definition  Res : Set := (list (Lab * Loc)).
Parameter stack : Var -> Res.
```

The environmental information of the stack is represented as a function associating a result to each (declared) variable. This map is never defined effectively: `(stack x v)` corresponds to $x \mapsto v$, which is discharged from the proof environment but never proved as a judgment. Correspondingly, assumptions about `stack` will be discharged in the rules in order to associate results to variables.

On the other hand, stores cannot be distributed in the proof environment. A store is a finite list of *method closures*; the i-th element of the list is the closure associated to ι_i. Closures are a body abstracted with respect to the $self$ variable; where closure bodies are an inductive datatype with a higher-order constructor similar to `let`:

```
Inductive Body     : Set := ground : Term -> Body
                          | bind   : Res -> (Var -> Body) -> Body.
Definition Closure : Set := (Var -> Body).

Definition Store : Set := (list Closure).
```

Some functions are needed for manipulating the structures we have introduced so far (e.g.,, for projecting single lists from lists of pairs, for generating new results from objects and results, etc.); see Appendix B.1 for the code.

Extra notions and judgments. We formalize `closed` by means of a function, in order to simplify the statement of the operational semantics and the proofs in `Coq`. The intended behaviour of this function, defined by mutual recursion on the structure of terms and objects, is to propagate in depth the predicate `closed`, thus reducing a predicate `(closed t)` into a predicate about simpler terms:

```
Parameter dummy : Var -> Prop.
Fixpoint closed [t:Term] : Prop := Cases t of
         (var x)      => (dummy x)
    |    (obj ml)     => (closed_obj ml)
    |    (over a l m) => (closed a) /\ ((x:Var) (dummy x) -> (closed (m x)))
    |    (call a l)   => (closed a)
    |    (clone a)    => (closed a)
    |    (lEt a b)    => (closed a) /\ ((x:Var) (dummy x) -> (closed (b x)))
with closed_obj [ml:Object] : Prop := Cases ml of
         (obj_nil)    => True
    |(obj_cons l m nl) => (closed_obj nl) /\ ((x:Var)(dummy x)->(closed (m x)))
end.
```

In the translation, we use locally quantified variables as placeholders for bound variables; thus they have not to be considered as "free" variables. We mark them as "dummy" by an auxiliary predicate `dummy`, where dummy variables are considered "closed". The proposition resulting from the `Simp`lification of a `(closed t)` goal is easily dealt with using the tactics provided by `Coq`. In the same way, we define also the functions `notin : Var -> Term -> Prop` and `fresh : Var -> (list Var) -> Prop`, which capture the "freshness" of a variable in a term and *w.r.t.* a list of variables, respectively (see Appendix B.2). Finally, the judgment `wrap` is formalized via an inductive predicate:

```
Inductive   wrap : Term -> Body -> Prop:=
         w_ground: (b:Term)
                   (closed b) -> (wrap b (ground b))
    |    w_bind  : (b:Var -> Term) (c:Var -> Body) (y:Var) (v:Res) (xl:Varlist)
                   ((z:Var) (dummy z) /\ (fresh z xl) -> (wrap (b z) (c z))) ->
```

```
(stack y) = (v) ->
((z:Var) ~(y=z) -> (notin y (b z))) ->
(wrap (b y) (bind v c)).
```

In the rule `w_bind`, the premise `((z:Var) ~(y=z) -> (notin y (b z)))` ensures that `b` is a "good context" for `y`, that is, `y` does not occur free in `b`. Thus, the replacement $b\{z/y\}$ or rule (*w_bind*) of Figure 6 can be implemented simply as the application `(b z)`, where `z` is a local (*i.e.*, fresh) variable.

Term reduction judgment. The operational semantics of $\mathsf{imp}\varsigma$ is easily encoded by two mutual inductive judgments

```
Mutual Inductive eval : Store -> Term -> Store -> Res -> Prop := ...
       with eval_body : Store -> Body -> Store -> Res -> Prop := ...
```

Most of their rules are encoded straightforwardly. In particular, the rules for variables and *let* enlighten how the proof environment is used to represent the stack:

```
e_var  : (s:Store)(x:Var)(v:Res) (stack x) = (v) -> (eval s x s v)
e_let  : (s,s',t:Store) (a:Term) (b:Var -> Term) (v,w:Res)
         (eval s a s' v) ->
         ((x:Var) (stack x)=(v) -> (eval s' (b x) t w)) ->
         (eval s (let a b) t w)
```

The formalization of the rule (*e_obj*) uses some functions for manipulating stores, closures and results:

```
e_obj  : (s:Store) (ml:Object) (cl:(list Closure)) (xl:Varlist)
         (scan (proj_meth_obj (ml)) (cl) (xl) (distinct (proj_lab_obj ml))) ->
         (eval s (obj ml) (alloc s cl) (new_res_obj ml (size s)))
```

The function `alloc` simply appends the new list of closures to the old store, follows the usual order on natural numbers. The function `new_res_obj` produces a new result, collecting method names of the given object and pairing them to new pointers to the store. The function `scan` returns a predicate which has to be proved for ensuring that the methods of the object have distinct names.

The *method selection* uses the extra predicate `eval_body` for evaluating closures:

```
e_call : (s,s',t:Store) (a:Term) (v,w:Res) (c:Closure) (l:Lab)
         (eval s a s' v) -> (In l (proj_lab_res v)) ->
         (store_nth (loc_in_res v l s') s') = (c) ->
         ((x:Var) (stack x) = (v) -> (eval_body s' (c x) t w)) ->
         (eval s (call a l) t w)
```

The evaluation of the body takes place in an environment where a local variable `x` denoting "self" is associated to (the value of) the receiver object itself. The predicate `eval_body` is defined straightforwardly.

4.3 Type system for terms

The encoding of the typing system for terms is not problematic. Similarly to the case of stacks, we model the *typing environment* with a function which will be used only for associating *object types* to variables.

```
Parameter typenv : Var -> TType.
```

Thus, judgments for typing terms and object bodies are two mutually defined inductive predicates:

```
Mutual Inductive type : Term -> TType -> Prop :=
    |     t_sub  : (a:Term) (A,B:TType)
                   (type a A) -> (subtype A B) -> (type a B)
    |     t_var  : (x:Var) (A:TType)
                   (wftype A) -> ((typenv x) = A) -> (type x A)
...
    |     t_obj  : (ml:Object) (A:TType)
                   (type_obj A (obj ml) A) ->  (type (obj ml) A)
    |     t_call : (a:Term) (l:Lab) (A,B:TType)
                   (type a A) ->
                   (In l (labels A)) -> (type_from_lab A l) = (B) ->
                   (type (call a l) B)
...
with type_obj : TType -> Term -> TType -> Prop := ...
```

where `subtype` represents the *sub* predicate. Due to lack of space, we omit here its encoding, which makes use of some auxiliary predicates for permutation and extension of lists representing object types.

4.4 Typing of results

Coinductive Result Typing for impς^{nov}. The coinductive system for result typing of Figure 8 are easily rendered in `Coq` by means of `Coinductive` predicates. We only point out that in the encoding of `cores`, we have to carry along the whole (result) type while we scan and type the components of results.

```
CoInductive cotype_body : Store -> Body -> TType -> Prop :=
            t_coground: (s:Store) (b:Term) (A:TType)
                        (type b A) -> (cotype_body s (ground b) A)
      |     t_cobind  : (s:Store) (b:Var -> Body) (A,B:TType) (v:Res)
                        (cores A s v A) ->
                        ((x:Var)(typenv x) = (A) -> (cotype_body s (b x) B)) ->
                        (cotype_body s (bind v b) B)
with cores  : TType -> Store -> Res -> TType -> Prop :=
            t_covoid  : (A:TType) (s:Store)
                        (cores A s (nil (Lab * Loc)) (mktype (nil (Lab * TType))))
      |     t_costep  : (A,B,C:TType) (s:Store) (v:Res) (i:Loc) (c:Closure)
                        (l:Lab) (pl:(list (Lab * TType)))
                        (cores C s v A) ->
                        (store_nth i s) = (c) -> (lt i (size s)) ->
                        ((x:Var) (typenv x) = (C) -> (cotype_body s (c x) B)) ->
                        (list_from_type A) = (pl) ->
                        ~(In l (labels A)) -> ~(In i (proj_loc_res v)) ->
                        (cores C s (cons (l,i) v) (mktype (cons (l,B) pl))).
```

Inductive Result Typing for impς. When we move to impς, we need to formalize store types:

```
Definition SType : Set := (list (TType * TType)).
```

Suitable functions for store type manipulation (construction and destruction) are thus easily defined; see Appendix B.4. The main judgments `res`, `type_body` are formalized as (non mutual) inductive predicates:

```
Inductive res : SType -> TType -> Res -> TType -> Prop :=
          t_void   : (S:SType) (A:TType)
                     (res S A (nil (Lab * Loc)) (mktype (nil (Lab * TType))))
    |     t_step   : (S:SType) (A,B,C:TType) (v:Res) (i:Loc)
                     (l:Lab) (pl:(list (Lab * TType)))
                     (res S A v B) ->
```

```
                    (stype_nth_1 i S) = (A) -> (stype_nth_2 i S) = (C) ->
                    (lt i (dim S)) ->
                    (wftype A) -> (type_from_lab A l)=C ->
                    (list_from_type B) = (pl) ->
                    (wftype C) -> ~(In l (labels B)) ->
                    ~(In i (proj_loc_res v)) ->
                    (res S A (cons (l,i) v) (mktype (cons (l,C) pl))).

Inductive type_body : SType -> Body -> TType -> Prop :=
....
```

5 Metatheory of impς in Coq

One of the main aims of the formalization presented in the previous section is to allow for the formal development of important properties of impς. In this section we discuss briefly the formal development on the uppermost important, yet delicate to prove, property of Subject Reduction. We stated Subject Reduction as Theorems 6 and 8 for impς^{nov} and impς, respectively, which are formalized as follows:

```
Theorem SR_nov : (s,t:Store) (a:Term) (v:Res)
             (eval s a t v) -> (A:TType) (type a A) ->
             ((x:Var)(w:Res)(stack x)=(w) /\ (typenv x)=C -> (cores s w C)) ->
             (EX B:TType | (cores t v B) /\ (sub B A)).

Theorem SR : (s,t:Store) (a:Term) (v:Res)
             (eval s a t v) -> (A:TType)(type a A) -> (S:SType)(comp S s) ->
             ((x:Var)(w:Res)(C:TType)(stack x)=(w)/\(typenv x)=C -> (res S w C)) ->
             (EX B:TType | (EX T:SType |
               (res T v B) /\ (ext T S) /\ (comp T t) /\ (sub B A))).
```

Despite that `cores` is coinductive and `res` is inductive, both theorems are proved by structural induction on the derivation of `(eval s a t v)`.

Before proving these theorems it is necessary to address all the aspects concerning concrete structures, as stores, store types, objects, object types and results. Thus, many technical lemmata about operational semantics, term typing and result typing has been formalized and proved. It turns out that these lemmata are relatively compact and easy to prove, in the case of coinductive encoding, because we have not to deal with store types. In particular, we have taken full advantage of the possibility of making coinductive proofs for the `cores` predicate, via the `Cofix` tactic.

On the other hand, in the inductive encoding of result typing, closer to the original setting [1], the development of the metatheory is quite more involved due to the handling of store types. It is important noticing that we are able to reuse some of the proofs developed for the coinductive encoding with a minimal effort. These proofs are those not requiring an explicit inspection on the structure of store types; in this case we have simply to convert potentially coinductive proofs carried out on derivations into proofs by induction on the structure of results. This re-usability of proofs witnesses the important fact that the present approach is quite modular.

However, some properties dealing with linear structures (such as stores) get much more involved when we consider also store types. In this case we cannot reuse part, neither follow the pattern, of the proofs of the simplified encoding. Instead, we have to develop different techniques, often more involved than every other one in the coinductive approach. This confirms that having explicit linear structures in judgments is very cumbersome, and therefore that the choice of delegating the stack and typing environment to the metalinguistic proof context reduces considerably the length and the complexity of proofs.

Another crucial aspect of our formalization is the use of higher-order abstract syntax. When reasoning on hypothetical premises with locally quantified variables, we gain for free the generation of new (meta)variables, but we do not know *a priori* whether two different variables x and y denote different or equal names. Moreover we do not know whether a given variable occurs free or not in

a term. In fact, reasoning about datatypes in higher-order abstract syntax is not fully supported in actual logical frameworks.

In order to get the extra expressive power we need, we extend the framework with the *Theory of Contexts* [20, 30]. This is a simple set of axioms which allow for a smooth manipulations of these notions related to names and variables. In particular, the Theory of Contexts allows for the generation of "fresh" variables, via the *unsaturation axiom:* "$\forall M.\ \exists x.\ x \notin M$". However, in our application this axiom has to be slightly modified in order to take into account of types. More precisely, we adopt the unsaturation axiom with two flavours. The first axiom corresponds to the case of using metavariables as *placeholders*. This is useful in conjunction with typing properties:

```
Axiom unsat : (A:TType)(xl:Varlist)(EX x | (dummy x)/\(fresh x xl)/\(typenv x)=A).
```

The second axiom reflects the perspective of using metavariables for *variables* of the object language: we assume the existence of fresh names to be associated both to results and their type. We have two versions of this axiom, depending on which implementation of result typing we consider:

```
Axiom unsat2_cores : (s:Store) (v:Res) (A:TType) (cores s v A) ->
               (xl:(list Var))(EX x | (fresh x xl)/\(stack x)=v/\(typenv x)=A).

Axiom unsat2_res : (S:SType) (v:Res) (A:TType) (res S v A) ->
               (xl:(list Var))(EX x | (fresh x xl)/\(stack x)=v/\(typenv x)=A).
```

Their intuitive meaning is that we can always associate a fresh variable to a given value and a type, provided they are consistent in a given store, or *w.r.t.* a given store type. In fact, `unsat2_cores` and `unsat2_res` correspond exactly to the rule (Store x Typing) in [1].

6 Conclusions and Further Work

In this paper, we have studied the formal development of the theory of object-based calculus with types and side effects, such as impς, in type-theory based proof assistants, such as Coq. In the encoding of the syntax and semantics of the calculus, we have tried to take most advantage of the features of CC$^{(\mathrm{Co})\mathrm{Ind}}$, the coinductive type theoretic logical framework underlying Coq. Therefore, we have developed an original presentation of impς, in the setting of Natural Deduction Semantics (NDS) and weak Higher Order Abstract Syntax (HOAS). This reformulation is interesting *per se*, since it allows for a simpler and smoother treatment of complex properties, such as Subject Reduction and Type Soundness. In fact, for a significant fragment of impς, we have been able to eliminate "store types", in favour of *coinductive* typing systems. The complete system has been encoded in Coq, and the fundamental property of Subject Reduction formally proved.

To our knowledge, this is the first development of the theory of an object-based language with side effects, in Coq, using NDS, HOAS and coinduction. Our experience leads us to affirm that this approach, is particularly well-suited with respect to the proof practice of Coq, also in the very stressful case of a calculus featuring objects, methods, dynamic lookup, types, subtypes, and imperative features. Moreover, this perspective give rise to original, and more easy to deal with, presentation of the very same language. In fact the absence of the explicit environmental structures in judgments has a direct impact on the structure of the proofs, thus reducing their complexity.

Future work. The formalization of impς is part of a larger project involving the study, definition and certified implementation of a typed class-based language (of the SmallTalk family) and of its intermediate object-based language (of the Self family) and their tools (interpreters, compilers). Interpreters will run on their own virtual machine or compiled on a (hopefully certified) virtual machines, such as the JVM.

The literature reports some experiments concerning these advanced goals: Bertot uses the Coq system for certifying a compiler for an imperative language [5], Strecker proves the correctness of a compiler from Java source language to Java bytecode in the proof environment Isabelle [31].

However, none of these works adopts higher-order abstract syntax for dealing with binders. In general, these formalizations represent variables either with first-order names, or with de Brujin indexes [18, 14]. It is well-known that these approaches have the drawback of a difficult handling of α-equivalence, replacement of variables, substitution, etc.—all problems which can be avoided in a `HOAS` setting.

Unfortunately, up to now `HOAS` techniques are not suitable for proof extraction; therefore, our interpreter can be only be executed on a computer through the `Coq` virtual machine. Actually the extraction mechanism under the form of `Caml`-code would probably oblige us to translate the `HOAS` encoding to one based on a first-order representation of names and binders. Nevertheless, we plan to use the formalization we have presented in this paper, for *certifying*, rather than extracting, compilers and interpreters. This task, which apparently is more difficult than the automatic program extraction, could benefit by the use off `NDS` and `HOAS`.

Acknowledgments. The authors are grateful to Yves Bertot, Joëlle Despeyroux, and Bernard Serpette for fruitful discussions and comments on early formalisations of $\mathsf{imp}\varsigma$.

References

[1] M. Abadi and L. Cardelli. *A theory of objects*. Springer-Verlag, 1996.

[2] H. Barendregt and T. Nipkow, editors. *Proceedings of TYPES'93*, volume 806 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[3] G. Barthe, P. Courtieu, G. Dufay, and S. M. de Sousa. Tool-assisted specification and verification of the JavaCard platform. In *Proc. of AMAST*, volume 2422 of *LNCS*, pages 41–59, 2002.

[4] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. M. de Sousa. A formal executable semantics of the JavaCard platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *LNCS*, pages 302–319. Springer-Verlag, 2001.

[5] Y. Bertot. A certified compiler for an imperative language. Technical Report RR-3488, INRIA, 1998.

[6] Y. Bertot. Formalizing a JVML verifier for initialization in a theorem prover. In *Proc. Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*. Springer-Verlag, 2001.

[7] R. Burstall and F. Honsell. Operational semantics in a natural deduction setting. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 185–214. Cambridge University Press, June 1990.

[8] L. Cardelli. Obliq: A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.

[9] A. Ciaffaglione. *Certified reasoning on Real Numbers and Objects in Co-inductive Type Theory*. PhD thesis, Dipartimento di Matematica e Informatica, Università di Udine, Italy and LORIA-INPL, Nancy, France, 2003.

[10] T. Coquand. Infinite objects in type theory. In Barendregt and Nipkow [2], pages 62–78.

[11] J. Despeyroux. Proof of translation in natural semantics. In *Proceedings of the First Conference on Logic in Computer Science*, pages 193–205. The Association for Computing Machinery, 1986.

[12] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order syntax in Coq. In *Proc. of TLCA'95*, volume 905 of *Lecture Notes in Computer Science*, Edinburgh, Apr. 1995. Springer-Verlag. Also appears as INRIA research report RR-2556, April 1995.

[13] K. Fisher, F. Honsell, and J. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1994.

[14] G. Gillard. A formalization of a concurrent object calculus up to alpha-conversion. In *Proc. of CADE*, Lecture Notes in Computer Science. Springer-Verlag, June 2000.

[15] E. Giménez. An application of co-inductive types in Coq: Verification of the Alternating Bit Protocol. In S. Berardi and M. Coppo, editors, *Proc. TYPES'95*, volume 1158 of *Lecture Notes in Computer Science*, pages 134–152, Turin, Mar. 1995. Springer-Verlag, 1996.

[16] E. Giménez. Codifying guarded recursion definitions with recursive schemes. In J. Smith, editor, *Proc. of TYPES'94*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59, Båstad, Sweden, June 1995. Springer-Verlag.

[17] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, Jan. 1993.

[18] D. Hirschkoff. Bisimulation proofs for the π-calculus in the Calculus of Constructions. In *Proc. TPHOL'97*, volume 1275 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[19] F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In *Proc. ICALP'01*, volume 2076 of *Lecture Notes in Computer Science*, pages 963–978. Springer-Verlag, 2001.

[20] F. Honsell, M. Miculan, and I. Scagnetto. π-calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.

[21] M. Huisman. *Reasoning about Java programs in higer order logic with PVS and Isabelle*. PhD thesis, Katholieke Universiteit Nijmegen, 2001.

[22] INRIA. *The Coq Proof Assistant*, 2002. `http://coq.inria.fr/doc/main.html` .

[23] G. Kahn. Natural Semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[24] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3), 2003.

[25] M. Miculan. The expressive power of structural operational semantics with explicit assumptions. In Barendregt and Nipkow [2], pages 292–320.

[26] M. Miculan. *Encoding Logical Theories of Programs*. PhD thesis, Dipartimento di Informatica, Università di Pisa, Italy, Mar. 1997.

[27] A. Momigliano, S. Ambler, and R. Crole. A comparison of formalizations of the meta-theory of a language with variable bindings in Isabelle. Technical Report 2001/07, University of Leicester, 2001.

[28] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proc. of ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988. The Association for Computing Machinery.

[29] J. Rothe, H. Tews, and B. Jacobs. The coalgebraic class specification language CCSL. Technical report, Dresden-Nijmegen, 2000.

[30] I. Scagnetto. *Reasoning about Names In Higher-Order Abstract Syntax*. PhD thesis, Dipartimento di Matematica e Informatica, Università di Udine, Italy, Mar. 2002.

[31] M. Strecker. Formal verification of a Java compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*, volume 2392, pages 63–77. Springer-Verlag, 2002.

[32] H. Tews. A case study in coalgebraic specification: memory management in the FIASCO microkernel. Technical report, TU Dresden, 2000.

[33] J. van den Berg, B. Jacobs, and E. Poll. Formal specification and verification of javacard's application identifier class. In *Proceedings of the Java Card 2000 Workshop*, number 2041 in LNCS, 2001.

A The Calculus of (Co)Inductive Constructions

The $CC^{(Co)Ind}$ type theory. $CC^{(Co)Ind}$ is an impredicative intuitionistic type theory, with dependent inductive and coinductive types. Formally, it is a system for deriving assertions of the shape $\Gamma \vdash_\Sigma M : T$, where Γ is a list of type assignments to variables, *i.e.* $x_1{:}T_1, \ldots, x_n{:}T_n$; Σ is the *signature* (*i.e.*, a list of typed constants); M is a λ-term and T is its type.

Using the *propositions-as-types, and λ-terms-as-proofs* isomorphism, $CC^{(Co)Ind}$ can be viewed as a system for representing assertions of a higher-order intuitionistic logic and their proofs. This implies that valid assertions correspond to *inhabited* types (*i.e.*, types for which there exists a *closed* term of that type), and moreover proof-checking corresponds to type-checking.

For lack of space, we will not describe in detail the rich language of terms and types of $CC^{(Co)Ind}$ or its properties. We will only point out the important property of $CC^{(Co)Ind}$, namely that checking whether a given term has a given type in $CC^{(Co)Ind}$ is decidable. This is the crucial property which makes it possible to use $CC^{(Co)Ind}$ as the core of a proof checker. We will now discuss briefly some features of $CC^{(Co)Ind}$. *Simple inductive types* can be defined as follows

```
Inductive id :  term := id_1 :  term_1 |...| id_n :  term_n.
```

The name `id` is the name of the inductively defined object, and `term` is its type. The constructors of `id` are `id_1`,...,`id_n`, whose types are `term_1`,...,`term_n`, respectively. For instance, the set of natural numbers is defined as

```
Inductive nat :  Set := O : nat | S : nat -> nat.
```

Types of constructors have to satisfy a *positivity condition*, which, roughly, requires that `ident` may occur only in strictly positive positions in the types of the arguments of `id_1`,...,`id_n`. This condition ensures the soundness of the definition; for further details, see [22]. For instance, the following definition is not accepted

```
Inductive D : Set := lam :  (D -> nat) -> D.
```

Inductive definitions automatically provide *induction* and *recursion* principles over the defined type. These principles state that elements of the type are only those built by the given constructors. For instance, the automatically generated induction principle for `nat` is the well-known Peano principle

```
nat_ind :  (P:nat->Prop)(P O) ->
((n:nat)(P n)->(P (S n))) -> (n:nat)(P n).
```

Objects of inductive types are well-founded, that is, they are always built by a finite unlimited number of constructors. *Coinductive* types arise by relaxing this condition: coinductive objects, in fact, can be *non-wellfounded*, in that they can have an infinite number of constructors in their structure. Hence, coinductive objects are specified by means of non-ending (but effective) processes of construction, expressed as "circular definitions." For example, the set of streams of natural numbers, is defined as

```
CoInductive Stream :  Set := seq :  nat -> Stream -> Stream.
```

and the stream of all zeros is given by

```
CoFixpoint allzeros :  Stream := (seq O allzeros).
```

Of course, since coinductive types are non-wellfounded, they do not have any induction principle. The only way for manipulating coinductive objects is by means of *case analysis* on the form of the outermost constructor. In order to ensure soundness of corecursive definitions, these have to

satisfy a *guardedness condition* [22,16,15]. Roughly, the constant being defined may appear in the defining equation only within an argument of some of its constructors. `Coq` forbids "short-circuit" definitions like

```
CoFixpoint X : Stream := X.
```

An interesting possibility arises in $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, in connection with the propositions-as-types paradigm, due to the fact that proofs are first-class objects. Coinductive predicates can be rendered as coinductive types, and then these are propositions which have infinitely long (or circular) proofs. The guardedness condition on the well-formedness of infinite objects allows to make sense of such infinitely regressing proof arguments. One can consistently assume his thesis as a hypothesis provided its applications appear in the proof only when *guarded* by a constructor of the corresponding type. This is the propositional version of the *guarded induction principle* introduced by Coquand and Giménez [10,16,15] for reasoning on coinductive objects, and it is the constructive counterpart of coinductive proofs.

The `Coq` proof assistant. `Coq` is an interactive proof assistant for the type theory $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, developed by the INRIA. For a complete description, we refer to [22]. More specifically, `Coq` is an editor for interactively searching for an inhabitant of a type, in a top-down fashion by applying tactics step-by-step, backtracking if needed, and for verifying correctness of typing judgements. `Coq`'s specification language, `Gallina`, allows to express the type theory $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ in pure ASCII text, as follows:

$\lambda x{:}M.N$ is written `[x:M]N` $\quad$ $\Pi x{:}M.N$ is written `(x:M)N`

$(M\ N)$ is written `(M N)` $\quad$ $M \to N$ is written `M -> N`

We will not give an independent syntax for $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, but we will use its `Gallina` formulation. A proof search starts by entering `Lemma id : goal`, where `goal` is the type representing the proposition to prove. At this point, `Coq` waits for commands from the user, in order to build the proof term which inhabits `goal` (*i.e.* the proof). To this end, `Coq` offers a rich set of *tactics*, *e.g.*, introduction and application of assumptions, application of rules and previously proved lemmata, elimination of inductive objects, inversion of (co)inductive hypotheses and so on. These tactics allow the user to proceed in his proof search much like he would do informally. At every step, the type checking algorithm ensures the soundness of the proof. When the proof term is completed, it can be saved (by the command `Qed`) for future applications.

B Coq Code

B.1 Auxiliary functions for term reduction

```
Fixpoint proj_lab_obj [ml:Object] : (list Lab) :=
         Cases ml of  obj_nil          => (nil Lab)
                  |  (obj_cons l m nl) => (cons l (proj_lab_obj nl))
         end.
Fixpoint proj_lab_res [rl:Res] : (list Lab) :=
         Cases rl of  nil                => (nil Lab)
                  |  (cons (pair l x) sl) => (cons l (proj_lab_res sl))
         end.
Fixpoint proj_meth_obj [ml:Object] : (list (Var -> Term)) :=
         Cases ml of  obj_nil          => (nil (Var -> Term))
                  |  (obj_cons l m nl) => (cons m (proj_meth_obj nl))
         end.
Fixpoint proj_loc_res [rl:Res] : (list Loc) :=
         Cases rl of  nil                => (nil Loc)
                  |  (cons (pair l x) sl) => (cons x (proj_loc_res sl))
         end.
Fixpoint new_res_obj [ml:Object] : nat -> Res := [n:nat]
         Cases ml of  obj_nil          => (nil (Lab * Loc))
                  |  (obj_cons l m nl) => (cons (l,n) (new_res_obj nl (S n)))
```

```
        end.
Fixpoint new_res_res [rl:Res] : nat -> Res := [n:nat]
        Cases rl of nil                => (nil (Lab * Loc))
                | (cons (pair l x) sl) => (cons (l,n) (new_res_res sl (S n)))
        end.
Definition store_nth [n:Loc; s:Store] : Closure := (nth n s void_closure).
Fixpoint store_to_list [il:(list Loc)] : Store -> (list Closure) := [s:Store]
        Cases il of  nil        => (nil Closure)
                |  (cons i jl) => (cons (store_nth i s) (store_to_list jl s))
        end.
Fixpoint loc_in_res [v:Res] : Lab -> Store -> Loc := [l:Lab; s:Store]
        Cases v of  nil               => (size s)
                |  (cons (pair k i) w) => Cases (eq_lab k l)
                                          of true  => i
                                          |  false => (loc_in_res w l s)
                                          end.
        end.
Fixpoint eq_lab [m,n:Lab] : bool := Cases m n of O O => true
                                              |  (S p) (S q) => (eq_lab p q)
                                              |  (S p) O     => false
                                              |  O (S q)     => false
                                    end.
Fixpoint distinct [ll:(list Lab)] : Prop :=
        Cases ll of nil => True
                |  (cons l kl) => ~(In l kl) /\ (distinct kl)
        end.
```

The function `loc_to_res` performs an intrinsecally partial operation, but has to be totally defined in Coq. It actually searches for the pointer associated to a certain label, and so it is designed in such a way that if a label has not been found, it returns a dangling pointer. Notice that such behaviour is simply seen as "unnormal" during the formal proofs. The definitions of the other functions are straightforward.

B.2 Freshness predicates

```
Fixpoint notin [y:Var; t:Term] : Prop := Cases t of
        (var x)           => ~(y=x)
    |   (obj ml)          => (notin_obj y ml)
    |   (over a l m)      => (notin y a) /\
                             ((x:Var) ~(y=x) -> (notin y (m x)))
    |   (call a l)        => (notin y a)
    |   (clone a)         => (notin y a)
    |   (lEt a b)         => (notin y a) /\
                             ((x:Var) ~(y=x) -> (notin y (b x))) end
with notin_obj [y:Var; ml:Object] : Prop := Cases ml of
         obj_nil          => True
    |   (obj_cons l m nl) => (notin_obj y nl) /\
                             ((x:Var) ~(y=x) -> (notin y (m x))) end.

Fixpoint fresh [x:Var; l:(list Var)] : Prop := Cases l of
        nil         => True
   |    (cons y yl) => ~(x=y) /\ (fresh x yl) end.
```

B.3 Auxiliary functions for term typing

We have to implement some functions for manipulating the structure of object-types. We need functions for projecting lists of labels from lists or object types (*i.e.* `proj_lab_list,labels`), for

assembling and disassembling types (`insert`,`list_from_type`) and searching for labels (*i.e.* `type_from_lab`):

```
Fixpoint proj_lab_list [pl:(list (Lab * TType))] : (list Lab) :=
         Cases pl of nil             => (nil Lab)
                  |  (cons (l,A) ql) => (cons l (proj_lab_list ql))
         end.
Definition labels : TType -> (list Lab) := [A:TType]
           Cases A of (mktype pl)    => (proj_lab_list pl)
           end.
Definition insert : (Lab * TType) -> TType -> TType :=
           [a:(Lab * TType)] [A:TType]
           Cases A of (mktype pl)    => (mktype (cons a pl))
           end.
Definition list_from_type : TType -> (list (Lab * TType)) := [A:TType]
           Cases A of (mktype pl)    => (pl)
           end.
Fixpoint type_from_lab_list [pl:(list (Lab * TType))] : Lab -> TType := [l:Lab]
         Cases pl of nil => (mktype (nil (Lab * TType)))
                  | (cons (k,A) ql)  => Cases (eq_lab k l)
                                        of true  => A
                                        |  false => (type_from_lab_list ql l)
                                        end
         end.
Definition type_from_lab : TType -> Lab -> TType := [A:TType] [l:Lab]
           Cases A of (mktype pl)    => (type_from_lab_list pl l)
           end.
```

B.4 Auxiliary functions for store type manipulation

We introduce store types and easily define projection functions on store types and functions for constructing store types from object types:

```
Definition SType : Set := (list (TType * TType)).

Definition stype_nth : Loc -> SType -> (TType * TType) := [n:Loc; S:SType]
           (nth n S ((mktype (nil (Lab * TType))), (mktype (nil (Lab * TType))))).

Definition stype_nth_1 : Loc -> SType -> TType := [n:Loc; S:SType]
           Cases (stype_nth n S) of (A,B) => A
           end.

Definition stype_nth_2 : Loc -> SType -> TType := [n:Loc; S:SType]
           Cases (stype_nth n S) of (A,B) => B
           end.

Fixpoint scan_type [pl:(list (Lab * TType))] : TType -> SType := [A:TType]
         Cases pl of nil => (nil (TType * TType))
                  |  (cons (l,B) ql) => (cons (A,B) (scan_type ql A))
         end.

Definition build_stype : TType -> SType := [A:TType]
           Cases A of (mktype pl) => (scan_type pl A)
           end.
```

C Subject Reduction for $\mathbf{imp}\varsigma^{\mathbf{nov}}$

Lemma 2 ***(Properties of the typing system)***
Let Γ be a well-formed context. Then:

$(var).$: $\forall x : Var;\ A : TType.$
$\Gamma \vdash type(x, A) \Rightarrow$
$\exists B : TType.\ (x \mapsto B \in \Gamma)\ \wedge\ \Gamma \vdash sub(B, A)$

$(obj).$: $\forall ml : Object;\ A : TType.$
$\Gamma \vdash type([l_i = \varsigma(x_i)b_i\ ^{i\in I}], A) \Rightarrow$
$\exists B : TType.\ \Gamma \vdash type([l_i = \varsigma(x_i)b_i\ ^{i\in I}], B)\ \wedge\ \Gamma \vdash (B <: A)$

$(call).$: $\forall a : Term;\ l : Lab;\ A, B : TType.$
$\Gamma \vdash type(a.l, A) \Rightarrow$
$\exists B, B_l : TType.\ \Gamma \vdash type(a, B)\ \wedge\ (l, B_l) \in B\ \wedge\ \Gamma \vdash sub(B_l, A)$

$(over).$: $\forall a : Term;\ m : Var \to Term;\ l : Lab;\ A : TType.$
$\Gamma \vdash type(a.l \leftarrow \varsigma(x)b, A) \Rightarrow$
$\exists B, B_l : TType.$
$\Gamma \vdash type(a, B)\ \wedge\ (l, B_l) \in B\ \wedge\ \Gamma \vdash sub(B, A)\ \wedge\ \Gamma, x \mapsto B \vdash type(b, B_l)$

$(clone).$: $\forall a : Term;\ A : TType.$
$\Gamma \vdash type(clone(a), A) \Rightarrow$
$\exists B : TType.\ \Gamma \vdash type(a, B)\ \wedge\ \Gamma \vdash sub(B, A)$

$(let).$: $\forall a : Term;\ b : Var \to Term;\ A, B : TType.$
$\Gamma \vdash type(let(a, \lambda x.b), A) \Rightarrow$
$\exists B, C : TType.\ \Gamma \vdash type(a, C)\ \wedge\ \Gamma \vdash sub(B, A)\ \wedge\ \Gamma, x \mapsto C \vdash type(b, B)$

$(sub{-}wt).$: $\forall A, B : TType.$
$\Gamma \vdash sub(A, B) \Rightarrow \Gamma \vdash (wt\ A)\ \wedge\ \Gamma \vdash (wt\ B)$

$(type{-}wt).$: $\forall a : Term;\ A : TType.$
$\Gamma \vdash type(a, A) \Rightarrow \Gamma \vdash (wt\ A)$

$(bound{-}weak).:$: $\forall a : Term;\ A, B, C : TType.$
$\Gamma, x \mapsto A \vdash type(b, C)\ \wedge\ sub(B, A) \Rightarrow$
$\Gamma, x \mapsto B \vdash type(b, C)$

Proof. All the properties are proved by structural induction. □

Some preventive work address the formal development of the metatheory about the result typing and specific properties of the various operators of the calculus. In the following Γ is a well-formed context that is omitted from judgments.

Lemma 3 ***(Co-inductive result typing)***

$(i).$: $(cores\ s\ v\ A)\ \wedge\ (eval\ s\ a\ t\ w) \Rightarrow$
$(cores\ t\ v\ A)$

$(ii).$: $\Gamma, x \mapsto A \vdash (type\ b\ B)\ \wedge\ \Gamma, closed(x) \vdash (wrap\ b\ c)\ \wedge\ s(\iota) = \lambda x.c\ \wedge$
$(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (cores\ s\ w\ C)) \Rightarrow$
$\Gamma, x \mapsto A \vdash (cotype_b\ s\ c\ B)$

Proof. (i). By structural induction on the derivation of $\Gamma \vdash (eval\ s\ a\ t\ w)$, because the store s cannot be updated.

(ii). By structural induction on the derivation of $\Gamma, closed(x) \vdash (wrap\ b\ c)$. □

Lemma 4 ***(Objects)***

$$\begin{array}{lll}(i). & : & (type\ [l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]\ [l_i : B_i\ ^{i\in 1..n}]) \ \wedge\ \Gamma, closed(x_i) \vdash (wrap\ b_i\ c_i)^{\,i\in 1..n} \wedge \\ & & (\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (cores\ s\ w\ C)) \ \Rightarrow \\ & & (cores\ (s, \iota_i \mapsto c_i\ ^{i\in 1..n})\ [l_i = \iota_i\ ^{i\in 1..n}]\ [l_i : B_i\ ^{i\in 1..n}])\end{array}$$

Proof. (i). By induction on the object $[l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]$ and lemma 3.(ii). □

Lemma 5 ***(Select)***

$$\begin{array}{lll}(i). & : & (cores\ s\ [l_j : \iota_j, \ldots]\ [l_j : B_j, \ldots]) \ \wedge\ s(\iota_j) = \lambda x.c \ \Rightarrow \\ & & \Gamma, x \mapsto [l_j : B_j, \ldots] \vdash (cotype_b\ s\ c\ B_j)\end{array}$$

Proof. (i). By induction on the object type $[l_j : B_j, \ldots]$. □

Lemma 6 ***(Clone)***

$$\begin{array}{lll}(i). & : & (cores\ s\ v\ A) \ \Rightarrow\ (cores\ (s,t)\ v\ A) \\ \\ (ii). & : & (cores\ s\ [l_i = \iota_i\ ^{i\in 1..n}]\ A) \ \Rightarrow \\ & & (cores\ (s, \iota'_i \mapsto s(\iota_i)\ ^{i\in 1..n})\ [l_i = \iota'_i\ ^{i\in 1..n}]\ A)\end{array}$$

Proof. (i). By co-induction.

(ii). By induction on the result $[l_i = \iota_i\ ^{i\in 1..n}]$. □

Theorem 9 ***(Subject Reduction, co-inductive setting, impς without Update)***
Let Γ be a well-formed context. Then:

$$\begin{array}{l}\Gamma \vdash (type\ a\ A) \ \wedge\ \Gamma \vdash (eval\ s\ a\ t\ v) \wedge \\ (\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (cores\ s\ w\ C)) \ \Rightarrow \\ \exists A^+ : TType. \\ \Gamma \vdash (cores\ t\ v\ A^+) \ \wedge\ \Gamma \vdash (sub\ A^+\ A)\end{array}$$

Proof. By structural induction on the derivation of $\Gamma \vdash (eval\ s\ a\ t\ v)$. The rules *e_call* and *e_bind* require a *mutual* structural induction argument, namely a stronger induction schema which holds for the predicate $eval_b$ as well, which is the counterpart of *eval* for closures. We present the proof by omitting the context Γ, that is, $\mathcal{J}$ stands for $\Gamma \vdash \mathcal{J}$.

(e_var). By hypothesis $(type\ x\ A)$ and:

$$\text{(e\_var)}\ \frac{x \mapsto v \in \Gamma}{(eval\ s\ x\ s\ v)}$$

From lemma 2.(var), there exists B such that $x \mapsto B \in \Gamma$ and $(sub\ B\ A)$. Choose $A^+ := B$.

Since $x \mapsto v \in \Gamma$, by the third hypothesis of the theorem we can derive $(cores\ s\ v\ A^+)$, thus concluding.

(e_obj). By hypothesis $(type\ [l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]\ A)$ and:

$$\text{(e\_obj)}\ \frac{\begin{array}{c}(closed(x_i)) \\ \vdots \\ (l_i, \iota_i\ distinct) \quad \iota_i \notin dom(\sigma) \quad (wrap\ b_i\ c_i) \quad \forall i \in 1..n\end{array}}{(eval\ s\ [l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]\ (s, \iota_i \mapsto \lambda x.c_i\ ^{i\in 1..n})\ [l_i = \iota_i\ ^{i\in 1..n}])}$$

By lemma 2.(obj), there exists $[l_i : B_i\ ^{i\in 1..n}]$ such that:

$$(type\ [l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]\ [l_i : B_i\ ^{i\in 1..n}]) \tag{1}$$

$$(sub\ [l_i : B_i\ ^{i\in 1..n}]\ A) \tag{2}$$

Choose $A^+ := [l_i : B_i\ ^{i\in 1..n}]$.

Since $\Gamma, closed(x_i) \vdash (wrap\ b_i\ c_i)\ \forall i \in 1..n$ and 1, we apply the lemma 4.(i), thus deducing $(cores\ (s, \iota_i \mapsto \lambda x_i.c_i\ ^{i\in 1..n})\ [l_i = \iota_i\ ^{i\in 1..n}]\ A^+)$; we conclude by 2.

(e_call). By hypothesis $(type\ (a.l_j)\ A)$ and:

$$\text{(e\_call)}\ \frac{(eval\ s\ a\ s'\ [l_i = \iota_i\ ^{i\in 1..n}])\quad j \in 1..n\quad s'(\iota_j) = \lambda x.c\qquad \begin{array}{c}(x \mapsto [l_i = \iota_i\ ^{i\in 1..n}])\\ \vdots \\ (eval_b\ s'\ c\ t\ w)\end{array}}{(eval\ s\ (a.l_j)\ t\ w)}$$

By lemma 2.(call), there exists $[l_j : B_j, \ldots]$ such that $(type\ a\ [l_j : B_j, \ldots])$ and $(sub\ B_j\ A)$. Since $(eval\ s\ a\ s'\ [l_i = \iota_i\ ^{i\in 1..n}])$, by inductive hypothesis there exists C such that:

(a). $(cores\ s'\ [l_i = \iota_i\ ^{i\in 1..n}]\ C)$;

(b). $(sub\ C\ [l_j : B_j, \ldots])$.

By the rule (e_call), it is $j \in 1..n$, $s'(\iota_j) = \lambda x.c$ and:

$$\Gamma, x \mapsto [l_i = \iota_i\ ^{i\in 1..n}] \vdash (eval_b\ s'\ c\ t\ w) \tag{3}$$

On the other hand, we have $C \equiv [l_j : B_j, \ldots]$ from (b), thus, using (a) and lemma 5.(i):

$$\Gamma, x \mapsto C \vdash (type_b\ s'\ c\ B_j) \tag{4}$$

We can deduce $(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (cores\ s'\ w\ C))$ from the third hypothesis of the theorem and lemma 3.(i); therefore, since 3 and 4, we can apply the mutual induction hypothesis, deducing there exists A^+ such that $(cores\ t\ w\ A^+)$ and $(sub\ A^+\ B_j)$. We finish by transitivity of subtyping.

(e_clone). By hypothesis $(type\ (clone\ a)\ A)$ and:

$$\text{(e\_clone)}\ \frac{(eval\ s\ a\ s'\ [l_i = \iota_i\ ^{i\in 1..n}])\quad \iota_i \in dom(s')\quad (\iota'_i\ distinct)\quad \iota'_i \notin dom(s')\quad \forall i \in 1..n}{(eval\ s\ (clone\ a)\ (s', \iota'_i \mapsto s'(\iota_i)\ ^{i\in 1..n})\ [l_i = \iota'_i\ ^{i\in 1..n}])}$$

By lemma 2.(clone), there exists B such that $(type\ a\ B)$ and $(sub\ B\ A)$. Since $(eval\ s\ a\ s'\ [l_i = \iota_i\ ^{i\in 1..n}])$, we can apply the inductive hypothesis, thus deducing there exists C such that:

(a). $(cores\ s'\ [l_i = \iota_i\ ^{i\in 1..n}]\ C)$;

(b). $(sub\ C\ B)$.

Choose $A^+ := C$. We deduce $(sub\ A^+\ A)$ by transitivity of subtyping. We conclude $(cores\ (s', \iota'_i \mapsto s'(\iota_i)^{i\in 1..n})\ [l_i = \iota'_i\ ^{i\in 1..n}]\ A^+)$ from (a) and lemma 6.(ii).

(e_let). By hypothesis $(type\ (let\ a\ \lambda x.b)\ A)$ and:

$$\text{(e\_let)}\ \frac{(eval\ s\ a\ s'\ v)\qquad \begin{array}{c}(x \mapsto v)\\ \vdots \\ (eval\ s'\ b\ t\ w)\end{array}}{(eval\ s\ (let\ a\ \lambda x.b)\ t\ w)}$$

By lemma 2.(let), there exist B, C such that $(type\ a\ C)$ and $\Gamma, x \mapsto C \vdash (type\ b\ B)$ and $(sub\ B\ A)$. Since $(eval\ s\ a\ s'\ v)$, by inductive hypothesis there exists D such that:

(a). $(cores\ s'\ v\ D)$;
(b). $(sub\ D\ C)$.
Since $\Gamma, x \mapsto C \vdash (type\ b\ B)$ and (b), we use lemma 2.(bd-weak) for deriving $\Gamma, x \mapsto D \vdash (type\ b\ B)$. Next we deduce $(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (cores\ s'\ w\ C))$ from the third hypothesis of the theorem and lemma 3.(i). Because of $\Gamma, x \mapsto v \vdash (eval\ s'\ b\ t\ w)$, we apply again the induction hypothesis, thus obtaining E such that $(cores\ t\ v\ E)$ and $(sub\ E\ B)$. Choose $A^+ := E$ and conclude by transitivity of subtyping.

(e_ground). By hypothesis $(cotype_b\ s\ (ground\ a)\ A)$ and:

$$(\text{e\_ground})\ \frac{(eval\ s\ a\ t\ v)}{(eval_b\ s\ (ground\ a)\ t\ v)}$$

The assertion $(cotype_b\ s\ (ground\ a)\ A)$ has to be derived by the rule $(t\_coground)$, namely from $(type\ a\ A)$: therefore, by induction, there exists A^+ such that $(cores\ t\ v\ A^+)$ and $(sub\ A^+\ A)$.

(e_bind). By hypothesis $(cotype_b\ s\ (bind\ v\ \lambda y.c)\ A)$ and:

$$(\text{e\_bind})\ \frac{\begin{array}{c}(y \mapsto v)\\ \vdots \\ (eval_b\ s\ c\ t\ w)\end{array}}{(eval_b\ s\ (bind\ v\ \lambda y.c)\ t\ w)}$$

The assertion $(cotype_b\ s\ (bind\ v\ \lambda y.c)\ A)$ has to be derived by the rule (t_cobind), namely there exists B such that $(cores\ s\ v\ B)$ and $\Gamma, y \mapsto B \vdash (cotype_b\ s\ c\ A)$. Therefore, by mutual induction, there exists A^+ such that $(cores\ t\ w\ A^+)$ and $(sub\ A^+\ A)$. □

D Subject Reduction for $\mathbf{imp}_\varsigma$

As for the co-inductive setting, we have to state a preliminary work about the result typing and specific properties of the various operators of the calculus. In the following Γ is a well-formed context that is omitted from judgments.

Lemma 7 ***(Inductive result typing)***

(i). : $\Gamma \vdash (ext\ \Sigma\ \Sigma)$

(ii). : $\Gamma \vdash (ext\ \Sigma''\ \Sigma')\ \wedge\ \Gamma \vdash (ext\ \Sigma'\ \Sigma)\ \Rightarrow$
$\Gamma \vdash (ext\ \Sigma''\ \Sigma)$

(iii). : $\Gamma \vdash (res\ \Sigma\ v\ A)\ \wedge\ \Gamma \vdash (ext\ \Sigma'\ \Sigma)\ \Rightarrow$
$\Gamma \vdash (res\ \Sigma'\ v\ A)$

(iv). : $\Gamma, x \mapsto A \vdash (type\ b\ B)\ \wedge\ \Gamma, closed(x) \vdash (wrap\ b\ c)\ \wedge\ s(\iota) = \lambda x.c\ \wedge$
$(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (res\ \Sigma\ w\ C))\ \Rightarrow$
$\Gamma, x \mapsto A \vdash (type_b\ \Sigma\ c\ B)$

Proof. (i), (ii). The proofs are immediate.
(iii). By structural induction on the derivation of $\Gamma \vdash (res\ \Sigma\ v\ A)$.
(iv). By structural induction on the derivation of $\Gamma, closed(x) \vdash (wrap\ b\ c)$. □

Lemma 8 ***(Objects)***

(i). : $A \equiv [l_i : B_i\,^{i\in 1..n}] \Rightarrow$
$(res\ (\Sigma, \iota_i \mapsto (A \Rightarrow B_i)\,^{i\in 1..n})\ [l_i = \iota_i\,^{i\in 1..n}]\ A)$

(ii). : $A \equiv [l_i : B_i\,^{i\in 1..n}] \wedge$
$(type\ [l_i = \varsigma(x_i)b_i\,^{i\in 1..n}]\ A)\ \wedge\ \Gamma, closed(x_i) \vdash (wrap\ b_i\ c_i)\,^{i\in 1..n}\ \wedge\ (comp\ \Sigma\ s)\ \Rightarrow$
$(comp\ (\Sigma, \iota_i \mapsto (A \Rightarrow B_i)\,^{i\in 1..n})\ (s, \iota_i \mapsto \lambda x_i.c_i\,^{i\in 1..n}))$

Proof. (i). By the rule (t_res).
(ii). By induction on the object type A. □

Lemma 9 ***(Select)***

(i). : $(comp\ \Sigma\ s)\ \wedge\ s(\iota_i) = \lambda x.c\ \Rightarrow$
$\Gamma, x \mapsto \Sigma_1(\iota_i) \vdash (type_b\ \Sigma\ c\ \Sigma_2(\iota_i))$

Proof. By the rule (t_comp). □

Lemma 10 ***(Update)***

(i). : $\Gamma, x \mapsto A \vdash (type\ b\ B)\ \wedge\ \Gamma, closed(x) \vdash (wrap\ b\ c)\ \wedge$
$(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (res\ \Sigma\ w\ C)) \Rightarrow$
$\Gamma, x \mapsto A \vdash (type_b\ \Sigma\ c\ B)$

(ii). : $\Gamma \vdash (comp\ \Sigma\ s)\ \wedge\ \Gamma, x \mapsto \Sigma_1(i) \vdash (type_b\ \Sigma\ c\ \Sigma_2(i))\ \Rightarrow$
$\Gamma \vdash (comp\ \Sigma\ (s.\iota_i \leftarrow \lambda x.c))$

Proof. (i). By lemma 7.(iv).
(ii). By the rule (t_comp) and point (i). □

Lemma 11 ***(Clone)***

(i). : $(res\ \Sigma\ [l_i = \iota_i\,^{i\in 1..n}]\ A)\ \Rightarrow$
$(res\ (\Sigma, \iota'_i \mapsto \Sigma(\iota_i)\,^{i\in 1..n})\ [l_i = \iota'_i\,^{i\in 1..n}]\ A)$

(ii). : $(comp\ \Sigma\ s)\ \Rightarrow$
$(comp\ (\Sigma, \iota'_i \mapsto \Sigma(\iota_i)\,^{i\in 1..n})\ (s, \iota'_i \mapsto s(\iota_i)\,^{i\in 1..n}))$

Proof. (i). By induction on the result $[l_i = \iota_i\,^{i\in 1..n}]$.
(ii). By induction on the store type fragment $\iota'_i \mapsto \Sigma(\iota_i)\,^{i\in 1..n}$. □

Theorem 10 ***(Subject Reduction, inductive setting, full impς)***
Let Γ be a well-formed context. Then:

$\Gamma \vdash (type\ a\ A)\ \wedge\ \Gamma \vdash (eval\ s\ a\ t\ v)\ \wedge\ \Gamma \vdash (comp\ \Sigma\ s)\ \wedge$
$(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (res\ \Sigma\ w\ C))\ \Rightarrow$
$\exists A^+ : TType, \Sigma^+ : SType.$
$\Gamma \vdash (res\ \Sigma^+\ v\ A^+)\ \wedge\ \Gamma \vdash (ext\ \Sigma^+\ \Sigma)\ \wedge\ \Gamma \vdash (comp\ \Sigma^+\ t)\ \wedge\ \Gamma \vdash (sub\ A^+\ A)$

Proof. By structural induction on the derivation of $\Gamma \vdash (eval\ s\ a\ t\ v)$. The rules e_call and e_bind require a *mutual* structural induction argument, namely a stronger induction schema which holds for the predicate $eval_b$ as well, which is the counterpart of $eval$ for closures. We present the proof by omitting the context Γ, that is, $\mathcal{J}$ stands for $\Gamma \vdash \mathcal{J}$.

(e_var). By hypothesis $(type\ x\ A)$ and:

$$\text{(e\_var)}\ \frac{x \mapsto v \in \Gamma}{(eval\ s\ x\ s\ v)}$$

From lemma 2.(var), there exists B such that $x \mapsto B \in \Gamma$ and $(sub\ B\ A)$. Choose $A^+ := B$ and $\Sigma^+ := \Sigma$.

Since $x \mapsto v \in \Gamma$, by the fourth hypothesis of the theorem we can derive $(res\ \Sigma^+\ v\ A^+)$. We have $(ext\ \Sigma^+\ \Sigma^+)$ by lemma 7.(i) and $(comp\ \Sigma^+\ s)$ by hypothesis, thus concluding.

(e_obj). By hypothesis $(type\ [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}]\ A)$ and:

$$\text{(e\_obj)}\ \frac{\begin{array}{c}(closed(x_i))\\ \vdots\\ (l_i, \iota_i\ distinct) \quad \iota_i \notin dom(\sigma) \quad (wrap\ b_i\ c_i) \quad \forall i \in 1..n\end{array}}{(eval\ s\ [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}]\ (s, \iota_i \mapsto \lambda x.c_i{}^{i\in 1..n})\ [l_i = \iota_i{}^{i\in 1..n}])}$$

By lemma 2.(obj), there exists $[l_i : B_i{}^{i\in 1..n}]$ such that:

$$(type\ [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}]\ [l_i : B_i{}^{i\in 1..n}]) \tag{5}$$

$$(sub\ [l_i : B_i{}^{i\in 1..n}]\ A) \tag{6}$$

Choose $A^+ := [l_i : B_i{}^{i\in 1..n}]$ and $\Sigma^+ := \Sigma, \iota_i \mapsto (A^+ \Rightarrow B_i)^{\,i\in 1..n}$.

We have $(res\ \Sigma^+\ [l_i = \iota_i{}^{i\in 1..n}]\ A^+)$ by lemma 8.(i) and it is immediate that $(ext\ \Sigma^+\ \Sigma)$. Next, since $(comp\ \Sigma\ s)$ and 5, we apply the lemma 8.(ii), thus deriving $(comp\ \Sigma^+\ (s, \iota_i \mapsto \lambda x_i.c_i{}^{i\in 1..n}))$. We finish by 6.

(e_call). By hypothesis $(type\ (a.l_j)\ A)$ and:

$$\text{(e\_call)}\ \frac{\begin{array}{c}(x \mapsto [l_i = \iota_i{}^{i\in 1..n}])\\ \vdots\\ (eval\ s\ a\ s'\ [l_i = \iota_i{}^{i\in 1..n}]) \quad j \in 1..n \quad s'(\iota_j) = \lambda x.c \qquad (eval_b\ s'\ c\ t\ w)\end{array}}{(eval\ s\ (a.l_j)\ t\ w)}$$

By lemma 2.(call), there exists $[l_j : B_j, \ldots]$ such that $(type\ a\ [l_j : B_j, \ldots])$ and $(sub\ B_j\ A)$. Since $(eval\ s\ a\ s'\ [l_i = \iota_i{}^{i\in 1..n}])$, by inductive hypothesis there exist C, Σ' such that:

(a). $(res\ \Sigma'\ [l_i = \iota_i{}^{i\in 1..n}]\ C)$;

(b). $(ext\ \Sigma'\ \Sigma)$;

(c). $(comp\ \Sigma'\ s')$;

(d). $(sub\ C\ [l_j : B_j, \ldots])$.

From the rule (e_call), it is $j \in 1..n$, $s'(\iota_j) = \lambda x.c$ and:

$$\Gamma, x \mapsto [l_i = \iota_i{}^{i\in 1..n}] \vdash (eval_b\ s'\ c\ t\ w) \tag{7}$$

On the other hand, we have $C \equiv [l_j : B_j, \ldots]$ from (d), thus $\Sigma'(\iota_j) = (C \Rightarrow B_j)$ and so, by (c) and lemma 9.(i):

$$\Gamma, x \mapsto C \vdash (type_b\ \Sigma'\ c\ B_j) \tag{8}$$

We deduce $(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (res\ \Sigma'\ w\ C))$ from the fourth hypothesis of the theorem and lemma 7.(iii); therefore, since 7, 8 and (c), we can apply the mutual induction hypothesis, thus concluding there exist A^+, Σ^+ such that:

(e). $(res\ \Sigma^+\ w\ A^+)$;

(f). $(ext\ \Sigma^+\ \Sigma')$;

(g). $(comp\ \Sigma^+\ t)$;
(h). $(sub\ A^+\ B_j)$.
We finish by transitivity of *ext* (lemma 7.(ii)) and transitivity of subtyping.

(e_over). By hypothesis $(type\ (a.l \leftarrow \varsigma(x)b)\ A)$ and:

$$\text{(e\_over)}\ \frac{\begin{array}{c}(closed(x))\\ \vdots\\ (eval\ s\ a\ s'\ [l_i = \iota_i\ ^{i\in 1..n}])\quad j \in 1..n\quad \iota_j \in dom(s')\quad (wrap\ b\ c)\end{array}}{(eval\ s\ (a.l \leftarrow \varsigma(x)b)\ (s'.\iota_j \leftarrow \lambda x.c)\ [l_i = \iota_i\ ^{i\in 1..n}])}$$

By lemma 2.(over), there exists $[l_j : B_j, \ldots]$ such that $(type\ a\ [l_j : B_j, \ldots])$, $(sub\ [l_j : B_j, \ldots]\ A)$ and $\Gamma, x \mapsto [l_j : B_j, \ldots] \vdash (type\ b\ B_j)$. Since $(eval\ s\ a\ s'\ [l_i = \iota_i\ ^{i\in 1..n}])$, we can apply the inductive hypothesis, thus deducing there exist C, Σ' such that:
(a). $(res\ \Sigma'\ [l_i = \iota_i\ ^{i\in 1..n}]\ C)$;
(b). $(ext\ \Sigma'\ \Sigma)$;
(c). $(comp\ \Sigma'\ s')$;
(d). $(sub\ C\ [l_j : B_j, \ldots])$, that is, $C \equiv [l_j : B_j, \ldots]$.
Choose $A^+ := C$ and $\Sigma^+ := \Sigma'$.

By lemma 2.(bd-weak) we obtain $\Gamma, x \mapsto C \vdash (type\ b\ B_j)$, that is, using (a) and $j \in 1..n$:

$$\Gamma, x \mapsto \Sigma_1^+(\iota_j) \vdash (type\ b\ \Sigma_2^+(\iota_j)) \tag{9}$$

We derive $(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (res\ \Sigma^+\ w\ C))$ from the fourth hypothesis of the theorem and lemma 7.(iii). Next, because of $\Gamma, closed(x) \vdash (wrap\ b\ c)$ and 9, by lemma 10.(i):

$$\Gamma, x \mapsto \Sigma_1^+(\iota_j) \vdash (type_b\ c\ \Sigma_2^+(\iota_j)) \tag{10}$$

Since (c) and 10, we apply the lemma 10.(ii), thus deriving $(comp\ \Sigma^+\ (s'.\iota_j \leftarrow \lambda x.c))$. We conclude by transitivity of subtyping.

(e_clone). By hypothesis $(type\ (clone\ a)\ A)$ and:

$$\text{(e\_clone)}\ \frac{(eval\ s\ a\ s'\ [l_i = \iota_i\ ^{i\in 1..n}])\quad \iota_i \in dom(s')\quad (\iota'_i\ distinct)\quad \iota'_i \notin dom(s')\quad \forall i \in 1..n}{(eval\ s\ (clone\ a)\ (s', \iota'_i \mapsto s'(\iota_i)\ ^{i\in 1..n})\ [l_i = \iota'_i\ ^{i\in 1..n}])}$$

By lemma 2.(clone), there exists B such that $(type\ a\ B)$ and $(sub\ B\ A)$. Since $(eval\ s\ a\ s'\ [l_i = \iota_i\ ^{i\in 1..n}])$, we can apply the inductive hypothesis, thus deducing there exist C, Σ' such that:
(a). $(res\ \Sigma'\ [l_i = \iota_i\ ^{i\in 1..n}]\ C)$;
(b). $(ext\ \Sigma'\ \Sigma)$;
(c). $(comp\ \Sigma'\ s')$;
(d). $(sub\ C\ B)$.
Choose $A^+ := C$ and $\Sigma^+ := \Sigma', \iota'_i \mapsto \Sigma'(\iota_i)\ ^{i\in 1..n}$.

We deduce $(ext\ \Sigma^+\ \Sigma)$ by transitivity of the *ext* relation; similarly $(sub\ A^+\ A)$ by transitivity of subtyping. Next we have $(res\ \Sigma^+\ [l_i = \iota'_i\ ^{i\in 1..n}]\ A^+)$ from (a) and lemma 11.(i) and finally $(comp\ \Sigma^+\ (s', \iota'_i \mapsto s'(\iota_i)\ ^{i\in 1..n}))$ using (c) and lemma 11.(ii).

(e_let). By hypothesis $(type\ (let\ a\ \lambda x.b)\ A)$ and:

$$\text{(e\_let)}\ \frac{\begin{array}{c}(x \mapsto v)\\ \vdots\\ (eval\ s\ a\ s'\ v)\quad (eval\ s'\ b\ t\ w)\end{array}}{(eval\ s\ (let\ a\ \lambda x.b)\ t\ w)}$$

By lemma 2.(let), there exist B, C such that $(type\ a\ C)$ and $\Gamma, x \mapsto C \vdash (type\ b\ B)$ and $(sub\ B\ A)$. Since $(eval\ s\ a\ s'\ v)$, by inductive hypothesis there exist D, Σ' such that:

(a). $(res\ \Sigma'\ v\ D)$;
(b). $(ext\ \Sigma'\ \Sigma)$;
(c). $(comp\ \Sigma'\ s')$;
(d). $(sub\ D\ C)$.

Since $\Gamma, x \mapsto C \vdash (type\ b\ B)$ and (b), we use lemma 2.(bd-weak) for deriving $\Gamma, x \mapsto D \vdash (type\ b\ B)$. Next we deduce $(\forall x, w, C.\ (x \mapsto w, x \mapsto C \in \Gamma) \Rightarrow \Gamma \vdash (res\ \Sigma'\ w\ C))$ from (a) and lemma 7.(iii). Because of $\Gamma, x \mapsto v \vdash (eval\ s'\ b\ t\ w)$, we apply again the induction hypothesis, thus obtaining E, Σ'' such that:

(e). $(res\ \Sigma''\ w\ E)$;
(f). $(ext\ \Sigma''\ \Sigma')$;
(g). $(comp\ \Sigma''\ t)$;
(h). $(sub\ E\ B)$.

Choose $A^+ := E$ and $\Sigma^+ := \Sigma''$.

We conclude transitivity of *ext* and transitivity of subtyping. □

(e_ground). By hypothesis $(type_b\ \Sigma\ (ground\ a)\ A)$ and:

$$\text{(e\_ground)}\ \frac{(eval\ s\ a\ t\ v)}{(eval_b\ s\ (ground\ a)\ t\ v)}$$

The assertion $(type_b\ \Sigma\ (ground\ a)\ A)$ has to be derived by the rule (t_ground), namely from $(type\ a\ A)$; therefore, by induction, there exist A^+, Σ^+ such that $(res\ \Sigma^+\ v\ A^+)$, $(ext\ \Sigma^+\ \Sigma)$, $(comp\ \Sigma^+\ t)$ and $(sub\ A^+\ A)$.

(e_bind). By hypothesis $(type_b\ \Sigma\ (bind\ v\ \lambda y.c)\ A)$ and:

$$\text{(e\_bind)}\ \frac{\begin{array}{c}(y \mapsto v)\\ \vdots \\ (eval_b\ s\ c\ t\ w)\end{array}}{(eval_b\ s\ (bind\ v\ \lambda y.c)\ t\ w)}$$

The assertion $(type_b\ \Sigma\ (bind\ v\ \lambda y.c)\ A)$ has to be derived by rule (t_bind), namely there exists B such that $(res\ \Sigma\ v\ B)$ and $\Gamma, y \mapsto B \vdash (type_b\ \Sigma\ c\ A)$. Therefore, by mutual induction, there exist A^+, Σ^+ such that $(res\ t\ w\ A^+)$, $(ext\ \Sigma^+\ \Sigma)$, $(comp\ \Sigma^+\ t)$ and $(sub\ A^+\ A)$. □

Contents

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399